UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

GABRIEL BRONZATTI MORO

Uso das Características Computacionais de Regiões Paralelas OpenMP para Redução do Consumo de Energia

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Orientador: Prof. Dr. Lucas Mello Schnorr

Porto Alegre
2018

CIP — CATALOGAÇÃO NA PUBLICAÇÃO

Bronzatti Moro, Gabriel

Uso das Características Computacionais de Regiões Paralelas OpenMP para Redução do Consumo de Energia / Gabriel Bronzatti Moro. – Porto Alegre: PPGC da UFRGS, 2018.

69 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2018. Orientador: Lucas Mello Schnorr.

I. Mello Schnorr, Lucas. II. Título.

“Sei o que devo ser e ainda não sou, mas rendo graças a Deus por estar trabalhando, embora lentamente, por dentro de mim próprio, para chegar, um dia, a ser o que devo ser.”

— CHICO XAVIER

AGRADECIMENTOS

Primeiramente, dedico este trabalho a Deus que me fortaleceu em todos os momentos de dificuldade, devo a Ele graças por esta oportunidade. Também o dedico a minha família, minha mãe Jucéli, meu pai Pedro, minhas tias Jussara e Janaína, como toda minha família que sempre torceu pelo meu sucesso, auxiliando-me em todas as minhas necessidades. Da mesma forma, agradeço a minha namorada Thaíse por sempre estar ao meu lado, obrigado pelo apoio, paciência e compreensão. A memória das minhas avós Nelsinda, Iraci, Elizabeth e Maria, pois sem elas, não seria possível chegar a essa conquista. Também, quero trazer a memória da professora Marcia Cristina Cera, que sempre serviu de inspiração para mim e que me auxiliou para entrar nesta Universidade. Agradeço também a todos os meus amigos, a galera da sala 205 (Alef, Camelo, Arthur e o grande Vinicius), o meu grande amigo Matheus Serpa pelas dicas preciosas e todos aqueles que tive a oportunidade de conviver nesse tempo de estudo. Por final, em especial, agradeço ao meu orientador por todos os ensinamentos, agradeço a ele por não ter cessado em ajudar-me durante todo o trabalho e ter me motivado do início ao fim, para ele o meu respeito e admiração.

LISTA DE ABREVIATURAS E SIGLAS

CPU	Central Processing Unit
SO	Sistema Operacional
DCT	Dynamic Concurrency Throtling
DDCM	Dynamic Duty Clock Modulation
DVFS	Dynamic Voltage and Frequency Scaling
FPC	Flops Per Cycle
IPC	Instruction Per Cycle
IO	Input Output
LB	Load Balance
LPC	Loads Per Cycle
MPI	Message Passing Unterface
PE	Parallel Efficiency
SIMD	Single Instruction Multiple Data

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

Exploiting the Computing Characteristics of OpenMP Parallel Regions to Reduce Energy Consumption

ABSTRACT

Performance and energy consumption are fundamental requirements in computer systems. A very frequent challenge is to combine both aspects, searching to keep the high performance computing while consuming less energy. There are a lot of techniques to reduce energy consumption, but in general, they use modern processors resources or they require specific knowledge about application and platform used. In this work, we propose a performance analysis workflow strategy divided into two steps. In the first step, we analyze the parallel application behavior through the use of hardware counters that reflect CPU and memory usage. The goal is to obtain a per-region computing signature. The result of this first step is a configuration file that describes the duration of each region, their hardware counters, and source code identification. The second step runs the parallel application with different frequencies (low or high) according to the characterization obtained in the previous step. The results show a reduction of 1,89% in energy consumption for the Lulesh benchmark with an increase of 0,09% in runtime when we compare our approach against the governor Ondemand of the Linux Operating System.

Keywords: Energy Consumption, Parallel Applications, OpenMP.

RESUMO

Desempenho e consumo energético são requisitos fundamentais em sistemas de computação. Um desafio comumente encontrado é conciliar esses dois aspectos, buscando manter o mesmo desempenho, consumindo cada vez menos energia. Muitas técnicas possibilitam a redução do consumo de energia em aplicações paralelas, mas na maioria das vezes elas envolvem recursos encontrados apenas em processadores modernos ou um conhecimento amplo das características da aplicação e da plataforma alvo. Nesse trabalho propomos uma abordagem em formato de *Workflow*. Na primeira fase, o comportamento da aplicação paralela é investigado. A partir dessa investigação, a segunda fase realiza a execução da aplicação paralela com diferentes frequências (mínima e máxima) de processador, utilizando a caracterização das regiões, obtida na primeira fase da abordagem. Esse *Workflow* foi implementado em formato de biblioteca dinâmica, a fim de que ela possa ser utilizada em qualquer aplicação OpenMP. A biblioteca possui suporte as duas fases do *Workflow*, na primeira fase é gerado um arquivo que descreve as assinaturas comportamentais das regiões paralelas da aplicação. Esse arquivo é posteriormente utilizado na segunda fase, quando a biblioteca vai alterar dinamicamente a frequência de processador. O *benchmark* Lulesh é utilizado como cenário de testes da biblioteca, com isso o maior ganho obtido é a redução de 1,89% do consumo de energia. Esse ganho acarretou uma sobrecarga de 0,09% no tempo de execução. Ao comparar nossa técnica com a política de troca de frequência adotada pelo *governor* Ondemand do Sistema Operacional Linux, o ganho de 1,89% é significativo em relação ao *benchmark* utilizado, pois nele existem regiões paralelas de curta duração, o que impacta negativamente no *overhead* da operação de troca de frequência.

Palavras-chave: Consumo de Energia. Aplicações Paralelas. OpenMP.

SUMÁRIO

1 INTRODUÇÃO

Aplicações de alto desempenho são de extrema importância para a humanidade. Cada vez mais cientistas de diversas áreas necessitam de poder de computação para processar modelos complexos com o objetivo de avançar suas pesquisas. Além do desempenho dessas aplicações, o consumo de energia é uma preocupação constante em centros de supercomputadores, os quais fornecem máquinas potentes para executar aplicações que muitas vezes levam segundos, horas ou até mesmo dias em alguns casos. Nesse sentido, muitas pesquisas estão sendo realizadas para reduzir o consumo de energia tanto a nível de processador, trabalhando com diferentes configurações de frequência para o processamento (uso de *Dynamic Voltage and Frequency Scaling* - DVFS), como também a nível de software, utilizando técnicas de caracterização para conhecer melhor o comportamento de aplicações, a fim de realizar otimizações para redução do consumo de energia.

A técnica *Dynamic Voltage and Frequency Scaling* (DVFS) é muito utilizada em sistemas embarcados, *desktops* e também em *workstations* que executam aplicações de alto desempenho (SUEUR; HEISER, 2010a). Geralmente, essa técnica é utilizada em conjunto com metodologias que definem o momento exato para a troca de frequência ocorrer, dependendo do tipo de aplicação e da metodologia empregada, a técnica DVFS pode ser vantajosa tanto em termos de consumo de energia, quanto em desempenho. O *driver* CPUFreq é um exemplo de utilização da técnica DVFS no Sistema Operacional Linux. Ele fornece uma interface para alterar a frequência dos processadores e utilizar políticas pré-implementadas (BRODOWSKI, 2013). O *framework* disponibiliza as seguintes políticas (modos): *Performance* - usa a maior frequência disponível; *Powersave* - utiliza a mais baixa frequência; *Ondemand* - escolhe a frequência de acordo com o uso da CPU; *Conservative* - funciona como o *Ondemand*, mas o ajuste é mais refinado; e *Userspace* - utiliza a frequência informada pelo usuário ou por algum programa executado como super usuário (BRODOWSKI, 2013).

Cada aplicação possui um perfil comportamental de execução, o que está relacionado com o tamanho de entrada, suas estruturas de dados, a quantidade e o tipo de cálculos realizados, entre outros aspectos. Esse perfil pode ser utilizado para agrupar as aplicações de acordo com o componente computacional que limita o desempenho da aplicação. Sendo assim, podemos ter aplicações cujo desempenho é limitado pelos acessos à memória (*Memory-Bound*), outras que são limitadas pelo desempenho computacional do processador (*CPU-Bound*) e enfim aquelas limitadas pelas operações de entrada e saída (*IO-Bound*). Jesshope e Egan (2006) definem aplicações do tipo *Memory-Bound* como aquelas em que o desempenho aumenta a medida que for reduzido a taxa de *Cache misses* para o segundo nível de *Cache* L2. Já nas aplicações do tipo *CPU-Bound*, uma redução da taxa de *misses* não necessariamente se transformaria em um melhor desempenho, pois o "gargalo" é a espera pela CPU ou por IO (entrada/saída) e não a memória.

Embora seja possível classificar as aplicações nessas categorias, nem sempre uma aplicação terá um comportamento totalmente *Memory-Bound* ou *CPU-bound*. Na maioria dos casos, as regiões de código de uma mesma aplicação podem apresentar uma espera mais marcante pelo recurso de CPU, Memória ou IO. Aplicações como Lulesh por exemplo, possuem diferentes comportamentos em suas regiões paralelas. Algumas dessas regiões possuem uma taxa de *misses* na *Cache* L2 a baixo de 50%, como também existem regiões com IPC (*Instruction Per Cycle*) acima de 0,5. A partir de uma caracterização do comportamento dessa aplicação, seria possível então escolher regiões potenciais a serem executadas em uma frequência menor, com o objetivo de reduzir o consumo de energia.

Mas, como caracterizar as diferentes regiões paralelas de uma aplicação? Nesse cenário, a principal motivação do presente trabalho é alinhar a técnica de análise de contadores de hardware e a atribuição de diferentes frequências de processador por região paralela.

Dessa forma, o objetivo desse trabalho é propor uma metodologia de análise de desempenho que permita detectar o comportamento de regiões paralelas para reduzir o consumo de energia nas regiões mais limitadas pela Memória. É proposto nesse trabalho um *workflow* de análise, o qual é dividido em duas fases. Na primeira é proposta uma forma de identificar quais regiões da aplicação são *Memory-Bound*. Já a segunda fase atua sobre cada região identificada como *Memory-Bound*. Uma frequência específica por região é então utilizada durante a execução com a expectativa de reduzir o consumo de energia total. Além disso, apresentamos uma biblioteca capaz de realizar as duas fases do *workflow* proposto: instrumentando automaticamente a aplicação paralela, a fim de ajustar em cada região potencial, a menor frequência de processador. Diferente de outras abordagens, a biblioteca proposta não exige nenhuma instrumentação manual, permitindo que a coleta de dados das regiões paralelas seja realizada de forma automática, pois utiliza para isso a ferramenta Opari2 (LORENZ et al., 2014).

O texto deste documento está organizado da seguinte maneira:

- **Capítulo 2 (Conceitos Básicos)**: Neste capítulo são apresentados os conceitos básicos envolvidos nesse trabalho, tais como: as diferenças comportamentais de aplicações *Memory-Bound* e *CPU-Bound*; a diferença do consumo de energia do *package* e da memória principal; e as principais medidas para definir o comportamento de uma aplicação.
- **Capítulo 3 (Trabalhos Relacionados)**: Este capítulo apresenta o estado da arte da redução do consumo de energia de aplicações utilizando a técnica DVFS, tanto para aplicações distribuídas (MPI), aplicações paralelas de memória compartilhada (OpenMP) e aplicações híbridas.
- **Capítulo 4 (Workflow de Análise de Características Computacionais)**: Neste capítulo é apresentado nossa metodologia para identificar o comportamento *Memory-Bound* em aplicações paralelas, utilizando para isso, contadores de hardware específicos.
- **Capítulo 5 (LibNina: DVFS para Regiões Paralelas de Aplicações OpenMP)**: Este capítulo apresenta uma biblioteca que automatiza a primeira fase do *workflow* proposto e permite a execução da segunda fase. Para isso, a biblioteca coleta contadores de hardware utilizando PAPI e a partir disso utiliza a biblioteca `CPUFreq` para mudar a frequência de processador das regiões paralelas potenciais, durante a segunda fase.
- **Capítulo 6 (Validação e Resultados Experimentais)**: Neste capítulo são apresentados os resultados de validação da metodologia proposta, como também resultados de consumo de energia e tempo de execução obtidos com a biblioteca.
- **Capítulo 7 (Considerações Finais)**: Este capítulo apresenta uma visão geral do trabalho, como suas contribuições e trabalhos futuros.

2 CONCEITOS BÁSICOS

Este capítulo apresenta a fundamentação teórica necessária para a compreensão do trabalho. Na seção 2.1 é discutido o comportamento de aplicações, sendo elas limitadas pela memória ou pela CPU. A seção 2.2 apresenta a definição de balanceamento de carga, a diferença entre o balanceamento estático e dinâmico, como também o balanceamento realizado em aplicações OpenMP. Já na seção 2.3, os modelos de programação OpenMP *fork-join* e tarefas são abordados. A seção 2.4 introduz o uso de contadores de hardware para se obter métricas de desempenho. O capítulo é encerrado com a seção 2.5 que trata sobre o consumo de energia e apresenta a técnica DVFS (*Dynamic Voltage and Frequency Scaling*).

2.1 Comportamento de Aplicações

Diamond et al. (2011) descrevem que existem três padrões de aplicação em computação de alto desempenho: as aplicações regulares que possuem computações e acessos a memória, os quais são previsíveis; as irregulares que possuem acessos aleatórios a memória, como também conseguem alterar dinamicamente suas estruturas de dados; e as aplicações de grafos que utilizam acessos em diferentes localidades da memória, fazendo com que a localidade espacial da *Cache* seja a menor possível. Nessas diferentes classes de aplicações é possível encontrar regiões paralelas que possuem uma espera maior pela memória e outras que são limitadas mais pela CPU.

Ao realizar a leitura de uma determinada informação, a aplicação realiza buscas nas memórias de rascunho (*Cache*). Não encontrando a informação é necessário acessar a memória principal. O acesso a memória principal gera um custo maior de desempenho, visto que, a memória principal possui uma velocidade inferior à memória *Cache*, pois são tecnologias diferentes para fins diferentes. Nisso, a taxa de acertos dessa busca interfere diretamente no desempenho de aplicações *Memory-Bound*, pois quanto menos vezes for necessário acessar a memória principal, maior será o desempenho da aplicação. Nem sempre encontramos uma aplicação totalmente *Memory-Bound*. O que ocorre é que, aplicações desse tipo possuem uma grande parte de sua execução ociosa, interferindo no desempenho da aplicação como um todo (SUBRAMANIAN, 2015). Geralmente esse comportamento é encontrado em aplicações SIMD (*Single Instruction Multiple Data*), tais como processamento de imagens, vídeos e assim por diante.

Diferente das aplicações *Memory-Bound*, as aplicações *CPU-Bound* são limitadas pela espera por CPU. Essas aplicações são aquelas que possuem o desempenho totalmente afetado pela velocidade em que o processador realiza as operações aritméticas solicitadas (HUTCHESON; NATOLI, 2011). Um bom exemplo deste tipo de aplicação é a sequência de Fibonacci, a qual determina para dada posição, o seu respectivo valor na sequência de Fibonacci. Como esse valor depende da soma de seus dois predecessores na sequência, o algoritmo precisa calcular toda a série até esses números, a fim de obter o resultado para o valor de entrada. Implementações da série de Fibonacci que utilizam recursividade tendem a consumir o recurso CPU de forma intensa, fazendo com que ocorra a computação de várias instruções por ciclo de processamento.

2.2 Balanceamento de Carga

Li e Lan (2005) definem balanceamento de carga como uma ação que permite dividir a carga de trabalho da aplicação para vários processadores, máquinas ou *threads*, a fim de obter o aumento de desempenho em uma aplicação paralela. A carga de trabalho nesse contexto pode ser a divisão de dados ou tarefas, o que está relacionado diretamente com o tipo de aplicação a ser paralelizada. Por exemplo, em aplicações de tipo *Stencil*, as quais realizam o processo de convolução em determinadas regiões da matriz para obter o resultado de um novo elemento, o balanceamento de carga pode ser a divisão da matriz base utilizada pela aplicação. Dessa maneira, cada *thread* poderia atuar sobre determinadas regiões da mesma matriz base. Outra abordagem de balanceamento de carga para uma aplicação *Stencil* poderia ser a divisão das iterações realizadas sobre determinado elemento da matriz.

O balanceamento de carga estático leva em consideração um conhecimento prévio da plataforma de execução (características da máquina) e da aplicação. A divisão da carga de trabalho ocorre antes da computação, uma única vez, fazendo com que esse tipo de técnica possua uma sobrecarga menor, quando comparada com o balanceamento dinâmico (LI; LAN, 2005). Em contrapartida, no balanceamento de carga dinâmico a sobrecarga é maior, visto que, a divisão de carga precisa ser recalculada em vários momentos em tempo de execução.

Em aplicações de memória compartilhada é possível informar ao OpenMP o tipo de balanceamento que será realizado em determinada região da aplicação. No Código 2.1 é possível visualizar um exemplo de algoritmo de multiplicação de matrizes que utiliza um balanceamento estático. Como a carga é conhecida, nesse problema é possível obter um bom desempenho com esse tipo de escalonador, pois cada *thread* receberá a mesma quantidade de iterações a serem realizadas sobre partes da matriz. O acesso das *threads* aos pedaços da matriz é controlado pelas variáveis privadas e públicas definidas na diretiva OpenMP.

Código 2.1 – Uso de balanceamento estático em OpenMP.

```
#pragma omp parallel for private(i,j,k,tmp) schedule(static)
for(i=0;i < size; i++) {
  for(j=0;j < size; j++) {
    tmp=0;
    for(k=0; k < size; k++)
      tmp = tmp + A[i][k] * B[k][j];
    R[i][j] = tmp;
  }
}
```

Diferente do balanceamento estático, no balanceamento dinâmico a carga é heterogênea, fazendo com que algumas *threads* trabalhem mais do que as outras. Isso exige um comportamento adaptativo do escalonador, pois ele decidirá em tempo de execução, com base nas mudanças da aplicação e da plataforma de execução, qual é a melhor opção de balanceamento para melhorar o desempenho da aplicação (LI; LAN, 2005).

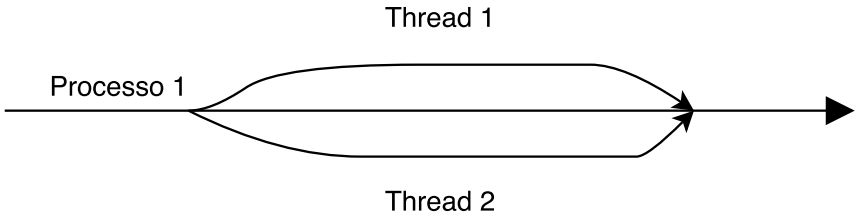
O conjunto de Mandelbrot é um exemplo de algoritmo de carga irregular. Nesse contexto, o balanceamento dinâmico é mais indicado do que o estático, pois a carga da aplicação é desconhecida. O algoritmo de Mandelbrot consiste no cálculo de quais pontos do plano fazem parte do conjunto de Mandelbrot, em cada ponto do plano (imagem) são realizados vários cálculos. Nesse algoritmo, a quantidade de iterações executadas

depende da localização do ponto sendo calculado nas coordenadas de Mandelbrot. Logo, isso gera uma carga desbalanceada entre as *threads* (CHANDRA, 2001).

2.3 Aplicações Paralelas em OpenMP para Memória Compartilhada

Existem vários modelos de aplicações paralelas de memória compartilhada, dentre eles, o mais comum em aplicações OpenMP é o *fork-join* (JANSEN, 2015). A Figura 2.1 apresenta um exemplo para ilustrar essa abordagem.

Figura 2.1: Modelo de Programação *fork-join*.



O *Fork-join* consiste em um processo que dispara várias *threads*. Após essa divisão, geralmente ocorre um processamento específico em cada *thread* ou um processamento similar, sobre diferentes dados (PACHECO, 2011). Essa técnica é muito utilizada na paralelização de laços "for". Em OpenMP, por exemplo, uma diretiva "pragma" indica ao compilador o local em que iniciará a divisão do fluxo principal de execução em vários outros fluxos de processamento (Figura 2.1). A seguir no código 2.2 é possível visualizar um exemplo do modelo de programação que utiliza a biblioteca OpenMP em um problema de multiplicação de matrizes.

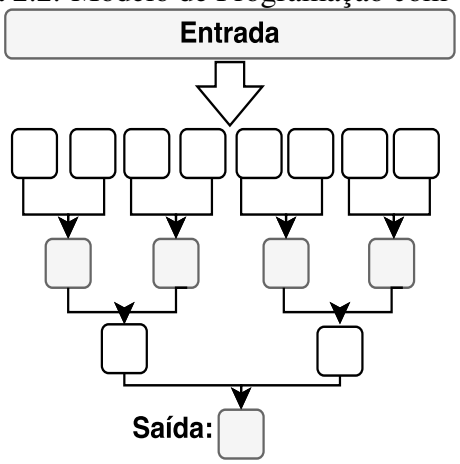
Código 2.2 – Multiplicação de matrizes usando *Fork-join* (KRAUSE; MORO; SCHNORR, 2016).

```
int i,j,k;
double tmp=0.0;

#pragma omp parallel for private(i,j,k)
for(i=0;i < size; i++) {
  for(j=0;j < size; j++) {
    tmp=0;
    for(k=0; k < size; k++)
      tmp = tmp + A[i * size + k] * B[k * size + j];
    R[i * size + j] = tmp;
  }
}
```

A abordagem tradicional de multiplicação de matrizes utiliza três laços aninhados, o primeiro que percorre as linhas da matriz, o mais interno permite o deslocamento nas colunas e o terceiro laço permite percorrer cada elemento (linha e coluna). Na implementação do Código 2.2, a diretiva "pragma omp parallel for" sinaliza o momento em que ocorrerá o disparo das *threads*, cada *thread* receberá do processo principal "x" iterações do primeiro laço, ou seja, a tarefa de percorrer "x" linhas da matriz. Cada *thread* executará o mesmo código, mas o índice "i" será diferente para cada *thread*, a fim de que elas realizem o mesmo trabalho em diferentes localizações da matriz.

Figura 2.2: Modelo de Programação com Tarefas.



Além do *Fork-join*, o modelo de tarefas é muito utilizado em OpenMP, ao construir programas que combinam várias soluções intermediárias para obter um resultado final. A saída desses programas é um resultado único, o qual foi calculado por processamentos consecutivos. Na Figura 2.2 é apresentado uma ilustração do modelo, onde a entrada é dividida em múltiplas tarefas que são então processadas par a par até obter a saída.

O algoritmo de Fibonacci é um exemplo de algoritmo que utiliza tarefas, essa implementação pode ser visualizada no Código 2.3. Esse algoritmo apresenta uma abordagem recursiva, na qual as duas soluções obtidas a cada etapa de recursão são somadas em uma apenas. O recurso da diretiva "omp task" permite realizar uma concorrência daquela tarefa entre as *threads*, as quais realizam o trabalho e após isso concorrem pela próxima tarefa. A diretiva de sincronização é simbolizada com o "pragma omp taskwait", ela define que as *threads* que terminaram as tarefas responsáveis por "x" e "y", esperem uma pela outra para realizar o cálculo de "x + y".

Código 2.3 – Fibonacci implementado usando tarefas OpenMP (ADDISON et al., 2009).

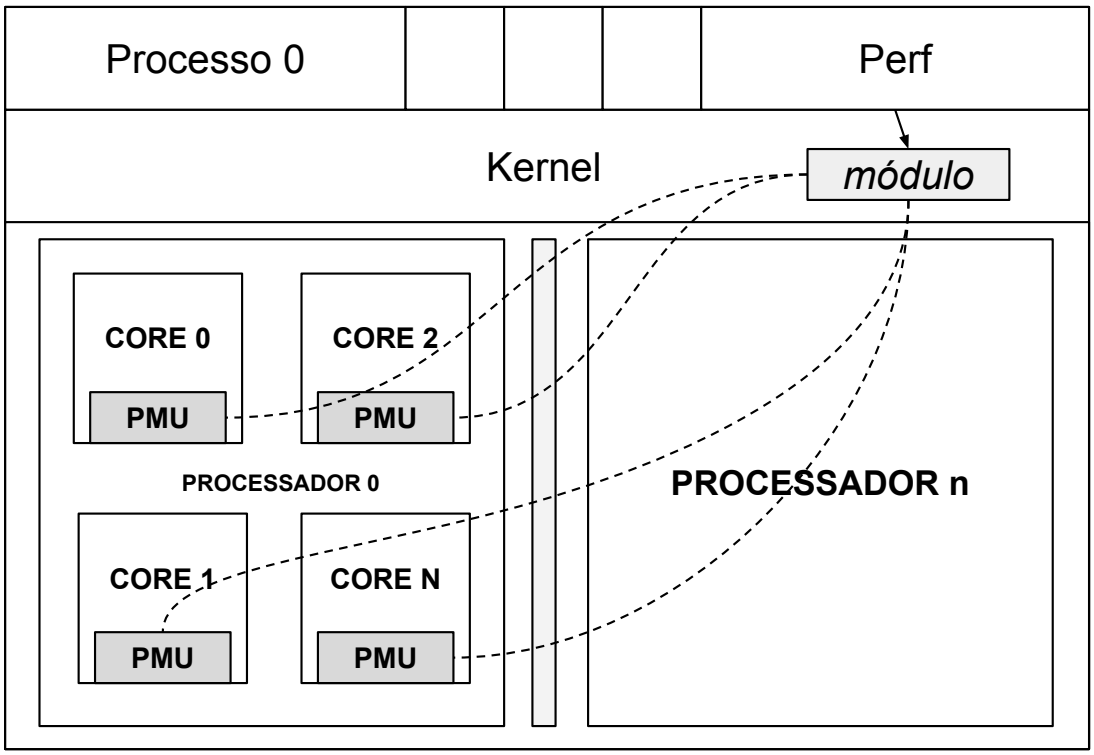
```
int fib(int n) {
    int x, y;
    if (n < 2)
        return n;
    else {
        #pragma omp task shared(x)
            x = fib(n - 1);
        #pragma omp task shared(y)
            y = fib(n - 2);
        #pragma omp taskwait
            return x + y;
    }
}
```

2.4 Métricas de Desempenho

Geralmente, cada núcleo de processamento possui unidades chamadas de *Performance Monitoring Units* (PMUs) que fornecem informações sobre o comportamento do sistema, utilizando contadores de eventos que registram a quantidade de ciclos, instruções, acessos a memória, entre outros eventos (MOLKA et al., 2017). A Figura 2.3 apresenta

um esquema para ilustrar a localização dessas unidades.

Figura 2.3: Esquema de localização das Unidades PMU



Cada PMU utiliza dois registradores, um destinado ao controle de qual informação contabilizar e outro para servir de acumulador. Essas informações podem ser acessadas utilizando a ferramenta PAPI ou Perf. A partir disso, as informações obtidas podem ser combinadas (métricas), a fim de se tornarem indicativos de desempenho. Nem sempre a plataforma fornecerá todos os eventos disponíveis.

As principais métricas para avaliar o desempenho de aplicações paralelas são *Flops Per Cycle* (FPC), *Loads Per Cycle* (LPC) e *Instructions Per Cycle* (IPC) (DIAMOND et al., 2011). A FPC define a taxa de computação algébrica realizada pela aplicação, esse indicativo é ótimo para medir a quantidade de cálculos com ponto flutuante realizados por ciclo de processador. O LPC permite analisar a quantidade de requisições *load* por ciclo, essa taxa permite verificar se o sistema de memória está colaborando com a CPU. Ela apresenta uma visão aproximada de possíveis *misses* que podem estar ocorrendo na aplicação. Já o IPC apresenta a taxa de instruções por ciclo, essa medida mapeia também os eventuais problemas de desempenho, como por exemplo as dependências entre as instruções.

Além das métricas de CPU, as métricas de memória são utilizadas para mapear a taxa de acessos a memória principal, como também a taxa de *Cache Misses* nos diferentes níveis de *Cache*. Essas métricas são muito utilizadas ao construir algoritmos, pois a partir delas é possível encontrar pontos a serem otimizados, levando em consideração a maneira em que os dados são armazenados e acessados na memória. Kowarschik e Weiß (2003) apresentam inúmeras técnicas para a diminuição da taxa de *misses* em aplicações, otimizando a localidade temporal e espacial de *Cache* nas aplicações.

2.5 Consumo de Energia

Segundo Orgerie, Assuncao e Lefevre (2014), o consumo de energia de recursos computacionais pode ser medido através de sensores de energia (gasto real) ou estimado a partir de modelos (estimativa teórica). Algumas ferramentas podem ser utilizadas para acessar esses sensores. Um exemplo de ferramenta que permite obter uma relação do consumo de energia total gasto pela aplicação é a Intel PCM. Essa ferramenta fornece o consumo de energia gasto pela CPU (considerando as memórias *Cache*) e memória principal (SILVEIRA et al., 2016). Outra maneira é estimar a energia utilizando modelos. CACTI é um exemplo de ferramenta que permite especificar o modelo de memória utilizado pela plataforma de execução, a partir disso é possível obter uma estimativa de gasto de energia por instrução (SILVEIRA et al., 2016).

Além de conhecer o quanto de energia a aplicação consome, vários trabalhos possuem propostas de redução no consumo energético. Dentre os assuntos investigados, uma técnica amplamente utilizada para economizar energia a partir da redução da frequência do processador é a técnica DVFS (*Dynamic Voltage and Frequency Scaling*). Essa técnica permite reduzir a frequência do processador em determinados trechos de execução do programa, os quais possuem um comportamento mais *Memory-Bound* (SUEUR; HEISER, 2010b). A seguir na equação retirada de Sueur e Heiser (2010b) é possível visualizar o impacto que a redução de frequência possui no cálculo de potência.

$$P = CfV^2 + P_{static} \tag{2.1}$$

O cálculo da potência (“P”) leva em consideração a capacitância (“C”) do circuito, a frequência em que as operações são realizadas e a voltagem (“V”) utilizada, ambos parâmetros são somados a potência estática. Assim é obtido a potência de sistema. A frequência está relacionada a velocidade em que as operações são realizadas, reduzindo esse valor é possível realizar a mesma carga de trabalho, só que o tempo de processamento será maior do que quando utilizado uma frequência maior. É fundamental conhecer essa relação, visto que, em trechos onde o processador espera por acessos à memória, reduzindo a frequência é possível reduzir o consumo de energia, sem impactar no desempenho da aplicação.

A técnica DVFS é um conjunto de técnicas que permitem mudar a frequência do processador para reduzir o consumo de energia em aplicações, sendo essas paralelas ou sequenciais (SUEUR; HEISER, 2010a). Ela pode ser utilizada com o *driver* CPUFreq do Sistema Operacional Linux, o qual fornece uma interface para alterar a frequência de processador utilizando diferentes formas de gerenciamento. O *driver* CPUFreq disponibiliza as seguintes políticas:

- *Performance* - utiliza a maior frequência disponível no processador;
- *Powersave* - utiliza a frequência mais baixa;
- *Ondemand* - utiliza uma frequência específica localizada no intervalo entre máxima e a mínima, de acordo com o uso da CPU. Geralmente essa política utiliza uma frequência mais alta, em trechos ociosos a frequência tende a diminuir;
- *Conservative* - como a política *Ondemand*, essa política altera a frequência do processador de acordo com sua taxa de uso. Ao invés de alternar em frequências altas e baixas (como o *Ondemand*), essa política altera a frequência de forma gradual;
- *Userspace* - utiliza a frequência informada pelo usuário ou por algum programa executado com permissões suficientes para efetuar este tipo de operação (BRO-

DOWSKI, 2013).

Dentre essas políticas, a *Userspace* se destaca, pois dependendo da plataforma, é possível atribuir inúmeras frequências de processador em diversos pontos da execução de uma aplicação, utilizando para isso a biblioteca fornecida pelo CPUFreq. Dessa forma, é possível realizar chamadas na região de código desejável, candidata a ser executada em uma frequência específica.

2.6 Conclusão do Capítulo

Ao longo desse capítulo foi possível apresentar os principais padrões de aplicação, os quais são elementares no processo de caracterização de uma aplicação como *Memory-Bound* ou *CPU-Bound*. Também, nós apresentamos os principais tópicos de balanceamento de carga, com exemplos de balanceamento estático e dinâmico no cenário de aplicações OpenMP em memória compartilhada. Vale salientar que, a combinação de variáveis como padrão de acessos a memória, modelo de programação, balanceamento de carga e métricas obtidas pela plataforma alvo impactam diretamente na caracterização de uma aplicação paralela, principalmente ao investigar trechos de execução nos quais a aplicação possui limitações de desempenho.

Através do estudo da equação de potência é possível visualizar o impacto que a frequência possui no consumo de potência, o que torna o gerenciamento de frequências de processador uma boa alternativa ao elaborar abordagens para reduzir o consumo de energia de aplicações. Alinhado com o conhecimento do comportamento de aplicações, o uso da técnica DVFS pode permitir o gerenciamento de quais frequências de processador utilizar em determinados momentos em que a aplicação não necessita de desempenho e pode ser executada em frequências inferiores, a fim de obter ganhos em consumo de energia.

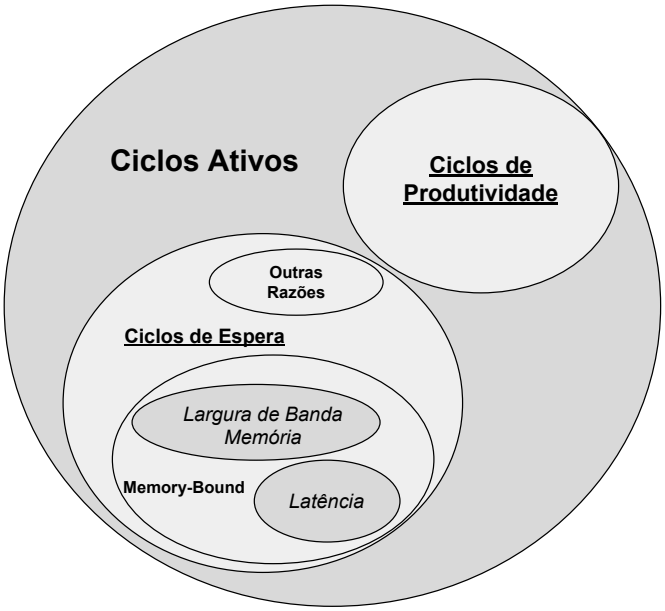
3 TRABALHOS RELACIONADOS

Neste capítulo são apresentados os principais trabalhos coletados que investigam o ajuste de frequência para aplicações paralelas e distribuídas. Esses trabalhos abordam o ajuste de frequência de processador levando em consideração fatores como: métricas de hardware, plataforma de execução e balanceamento de carga de trabalho entre os recursos de processamento.

3.1 Aplicações de Memória Compartilhada

Molka et al. (2017) apresentam uma abordagem que investiga quais contadores de hardware devem ser utilizados para se obter uma melhor estimativa do uso de componentes da hierarquia de memória. Para isso, os autores utilizam um conjunto de *benchmarks* específicos, os quais tem por objetivo o "estresse" individual dos componentes do sistema de memória (*Caches* e Memória Principal). Os *benchmarks* utilizados no trabalho permitem configurações como: restrição para medir apenas uma única CPU, configuração de uso de apenas 50% da *Cache* L1, diminuição do tamanho do último nível de *Cache* para menos de 10 vezes, dimensionamento de *loads* e *stores*, entre outras configurações para melhor examinar os componentes de memória utilizando apenas um processador, como também recursos compartilhados no cenário de uma plataforma *multi-core*. A Figura 3.1 apresenta uma decomposição de ciclos de processador criada pelos autores.

Figura 3.1: Decomposição de ciclos proposta por Molka et al. (2017).



Os ciclos de processador são divididos primeiramente em dois grupos, os ciclos de produtividade que são aqueles de processamento útil de instruções e os ciclos de espera que existem em função da latência gerada pelo acesso a memória ou por outras razões. Nos ciclos *Memory-Bound*, os autores identificaram duas situações, a primeira quando os ciclos de processador eram limitados pela espera por memória e a segunda situação quando eles eram limitados pela largura de banda da memória. Os autores mapearam os ciclos ativos e os ciclos de espera em uma aplicação OpenMP do *benchmark* NAS, após isso foi realizado uma redução de frequência de processador nos momentos em que a

aplicação possui ciclos de espera, através disso foi possível reduzir o consumo de energia em cerca de 4,16% com uma penalidade de 0,2% no tempo de execução.

Millani e Schnorr (2016) investigam a relação entre tempo e energia ao adotar frequências de processador por região paralela. Primeiramente, os autores executam a técnica *Screening Parallel Code Regions* para identificar possíveis indicadores de tempo de execução e consumo de energia para todas as regiões do programa paralelo. Na fase de *Screening*, as regiões paralelas são assinaladas com duas possíveis frequências (baixa ou alta), as quais devem ser suportadas pela plataforma alvo. Os autores utilizam as técnicas ANOVA (Análise de Variância) e *Main Effects Analysis* para identificar as regiões paralelas potenciais, aquelas que possuem maior impacto no consumo de energia e menor impacto no tempo de execução. Dessa forma, em uma nova execução do programa, apenas as regiões específicas (potenciais) sofrerão a alteração de frequência de processador. Dentre os resultados obtidos, os autores conseguiram uma redução de aproximadamente 9,27% no consumo de energia para o *benchmark* MiniFE, sem perdas significativas de desempenho. Por outro lado, a abordagem é totalmente dependente de instrumentação de código, exigindo um conhecimento prévio da aplicação ao atribuir a frequência de processador para as regiões paralelas, sendo uma desvantagem em cenários em que a aplicação é completamente desconhecida pelo analista de desempenho.

Diferente disso, Wang et al. (2015) possuem uma abordagem voltada mais às técnicas de gerenciamento de potência disponibilizadas em hardware. Ao invés de utilizar DVFS para alterar a frequência de processador a ser utilizada, os autores utilizam CPU *Clock Modulation* para definir a frequência adequada para os diferentes laços da aplicação. Dessa forma, é possível visualizar ganhos ao utilizar execuções com múltiplas frequências que são selecionadas com CPU *Clock Modulation*. Ao entrar em cada laço do programa, a frequência de processador é reduzida e ao sair a mesma é aumentada. Segundo os autores, a técnica pode em alguns casos acarretar uma sobrecarga no tempo de execução, mas um ganho considerável no consumo de energia. O melhor resultado apresentado pelos autores é uma redução média de 8,6% no consumo de energia, com a penalidade inferior a 1,5% no tempo de execução. Por utilizar CPU *Clock Modulation*, a metodologia só pode ser aplicada em plataformas com processadores modernos, o que pode dificultar a utilização da abordagem.

Laurenzano et al. (2011) definem uma abordagem que permite selecionar frequências de processador adequadas por laço de repetição da aplicação. Na caracterização das aplicações, os autores utilizam a taxa de *hit* na *Cache* L1, L2 e L3; a taxa de operações com ponto flutuante para o número de operações de memória; e a taxa de operações com números inteiros por operações com ponto flutuante. Os autores formam uma base de conhecimento de acordo com inúmeras execuções, caracterizando o consumo de energia gasto de acordo com padrões de execução e frequências de processador executadas anteriormente, obtidas por rastro. Dessa maneira é possível buscar nesse conjunto de cobertura, qual a melhor frequência para determinado padrão de aplicação. O melhor caso apresentado pelos autores é uma redução de aproximadamente 10,6% no consumo de energia, utilizando determinada carga de trabalho para uma arquitetura de 32 cores AMD Opteron 8380. Vale salientar que, a abordagem utiliza uma base de conhecimento para escolher a melhor configuração de consumo de energia e tempo de execução, sendo necessário várias execuções com diferentes fatores, dificultando uma análise preliminar com poucas execuções.

Em contra partida, Shafik et al. (2015) apresentam uma abordagem para redução do consumo de energia utilizando políticas de hardware, tais como DVFS e *Dynamic*

Concurrency Throtling (DCT). A abordagem investiga o desempenho da parte sequencial e paralela do programa, gerando anotações que serão utilizadas como uma espécie de marcadores, os quais descrevem o desempenho do programa. Esses marcadores são posteriormente incorporados na biblioteca OpenMP, como modificações na libgomp. Através de estatísticas obtidas por contadores de hardware em intervalos regulares são aplicadas as técnicas DVFS e DCT, tanto na parte sequencial como na parte paralela da aplicação. Dentre os resultados, os autores apresentam uma redução de aproximadamente 17% no consumo de energia. Como a técnica proposta por Wang et al. (2015), a abordagem de Shafik et al. (2015) depende de recursos de hardware, os quais podem ser encontrados apenas em processadores mais recentes.

3.2 Aplicações Distribuídas

Padoin et al. (2017) apresentam balanceadores de carga alinhados a técnica DVFS para reduzir o consumo de energia. A abordagem utiliza um sistema de *runtime* implementado com Charm++ *parallel runtime system*. Os autores apresentam um *framework* que não necessita de instrumentação de código-fonte, assim a carga de dados é conhecida dinamicamente em tempo de execução. Dentre os resultados obtidos, o trabalho apresenta uma redução de potência de cerca de 7,5%, quando a técnica é aplicada em uma granularidade mais fina. Em outro cenário, utilizando uma granularidade mais grossa, os autores conseguem uma redução de 18,75% no consumo de energia.

Etinski et al. (2009) apresentam um algoritmo que aplica DVFS e a técnica CPU *over-clocking*. Esse algoritmo tem por objetivo balancear os tempos de computação, a fim de fornecer um tempo de computação menor do que o da aplicação original (sem as modificações propostas). É utilizado também frequências limites em um intervalo de 10% a 20% a cima do limite especificado pelo fabricante do processador. Durante o tempo de execução da aplicação MPI, as situações de alta taxa de balanceamento de carga são tratadas, nelas busca-se alternar entre as frequências dinamicamente, até chegar a um tempo médio de computação. Além disso, Etinski et al. (2009) apresentam uma caracterização de aplicações MPI utilizando métricas como LB (*Load Balance)* e a PE (*Parallel Efficiency)*. A seguir são apresentadas as equações:

$$LB = \frac{\sum_{k=1}^{N_{proc} * TotalExecutionTimek}}{N_{proc} * maxComputationTime} \tag{3.1}$$

$$PE = \frac{\sum_{Nproc}^{K=1}}{N_{proc} * TotalExecutionTime} \tag{3.2}$$

A LB analisa a carga de trabalho realizada pela aplicação, tendo como entrada o número de processos (*Nproc*) e seu tempo total de execução, o somatório desses resultados é dividido pelo maior tempo de execução multiplicado por todos processos, fazendo com que seja possível obter uma porcentagem aproximada da carga de trabalho estimada por cada processo. Já a métrica *PE* é utilizada para analisar a taxa de computação útil da aplicação. Segundo os autores é possível obter em média 60% de redução no consumo de energia, utilizando um balanceamento de carga adequado. Além disso, por utilizar frequências a cima do intervalo definido pelo fabricante, ao utilizar essa técnica, a temperatura de processador pode atingir valores altos, acarretados pelo *over-clocking*.

Na mesma linha de caracterização de cargas de trabalho, Huang e Feng (2009)

apresentam um algoritmo que reduz o consumo de energia de acordo com o conhecimento da carga de trabalho das diferentes *threads* de processamento. Ele ajusta dinamicamente a frequência de processador para reduzir a potência. Esse algoritmo divide a aplicação em inúmeros intervalos de curta duração para mapear o comportamento de um dado intervalo, a fim de que, no próximo intervalo seja atribuída uma frequência aproximada de acordo com a caracterização realizada para o intervalo anterior. A caracterização utilizada pela abordagem leva em consideração a quantidade de sincronizações de processos, os acessos de IO (*Input Output*), como também a memória e os momentos em que a CPU está em estado de espera. O cálculo de carga de trabalho utilizado pelos autores é apresentado a seguir.

$$\lambda = \frac{C_{On}}{C_{On} + T_{Off} * f_{max}} \tag{3.3}$$

Diferente de Etinski et al. (2009) que utilizam o tempo de computação, Huang e Feng (2009) apresentam no cálculo do balanceamento de carga, o número de ciclos de processador. Ao alterar a frequência de processador, existem ciclos em que a frequência ainda será a antiga, esses ciclos são mapeados por Huang e Feng (2009) no parâmetro C_{On} da equação. Além disso, os autores levam em consideração os demais ciclos (C_{Off}), o tempo de execução (T_{Off}) e a frequência máxima (f_{max}). Os resultados mostram uma redução de 11% no consumo de energia, com uma perda de desempenho de aproximadamente de 5%, o que pode estar relacionado com a sobrecarga que o algoritmo possui ao caracterizar as fases e alterar a frequência de processamento. Em aplicações que possuem fases heterogêneas, os resultados de ganho podem ser menores, pois a frequência atribuída para a próxima fase da aplicação é baseada na fase anterior.

Voltado mais a utilização das políticas implementadas em hardware, Bhalachandra et al. (2017) definem um *runtime* adaptável por núcleo, o qual realiza mudanças de frequência de processador, de acordo com a carga de trabalho da aplicação. Esse *runtime* utiliza técnicas como DVFS, DDCM (*Dynamic Duty Clock Modulation*) e DVFS com DDCM. A técnica DVFS é utilizada para alterar a frequência do processador, já o DDCM fornece o tempo de espera e o tempo de computação de cada núcleo, assim é possível mapear características comuns de aplicações, tais como o tempo de processamento e o tempo de espera por comunicação. Também, o DDCM permite a redução da velocidade dos núcleos que estão executando mais rápido, a fim de garantir um melhor balanceamento de carga entre os recursos de processamento. Os resultados apresentam uma redução média de 31% no consumo de potência para a melhor configuração testada.

3.3 Aplicações Híbridas

Li et al. (2013) apresentam um algoritmo que tem como base um modelo de previsão de desempenho dedicado para aplicações híbridas MPI/OpenMP. Os autores propõem um *runtime* para controlar a execução das regiões paralelas OpenMP, as chamadas MPI e a utilização das políticas DVFS e *Dynamic Concurrency Throttling* (DCT). A abordagem utiliza contadores de hardware de desempenho para conhecer o comportamento da aplicação, focando principalmente nos eventos de hardware que fornecem o consumo médio de memória por região. Os resultados apresentam uma redução média de 8,7% no consumo de energia, com perdas insignificantes de desempenho.

Soma-se a essa abordagem, o trabalho de Lim, Freeh e Lowenthal (2011) que apresentam um *runtime* para aplicações MPI, o qual reduz dinamicamente a frequência

de processador durante as regiões de comunicação. O sistema de *runtime* agrupa por proximidade as regiões paralelas, nas quais ocorrem algum tipo de comunicação MPI, denominando-as como "regiões reduzíveis". Essa regiões agrupam chamadas MPI que ocorrem entre intervalos de tempo similar, elas são propícias para serem executadas em uma frequência de processador menor do que as demais. Para conhecer essas regiões, o sistema de *runtime* utiliza um componente de treino, capaz de "aprender" sobre o comportamento da aplicação alvo, sem necessitar de rastros de execução ou de uma análise de código para encontrar essas regiões. Dessa forma, o analista de desempenho não precisa interferir em seu funcionamento e nem fornecer informações sobre o comportamento da aplicação. Além do componente de treinamento, o sistema de *runtime* utiliza um componente de deslocamento, o qual determina quando o programa entra e sai da "região reduzível" e quando ocorrerá o deslocamento para a próxima região. Os resultados apresentam uma redução média de 11% no consumo de energia para os 12 *benchmarks* utilizados, com uma penalidade média de 2% no tempo de execução.

3.4 Conclusão do Capítulo

Dentre os trabalhos analisados é perceptível que existem várias formas de caracterizar uma aplicação paralela, sendo pelo uso de contadores de hardware, medidas de tempo de computação/comunicação, intervalos de tempo específicos em que a CPU fica ociosa e assim por diante. Algumas abordagens utilizam recursos disponibilizados pelo próprio hardware, o que possibilita melhores resultados na maioria dos casos. A Tabela 3.1 apresenta um comparativo dos trabalhos coletados, apresentando os seus respectivos ganhos de energia (coluna de Energia) e sobrecarga no tempo de execução (coluna Tempo). Além disso, a tabela apresenta um demonstrativo de quais técnicas os trabalhos utilizam, destacando abordagens que são encontradas em processadores recentes (coluna Recursos Esp.), outros que utilizam contadores de hardware, DVFS, a técnica de *Screening*, *Full Factorial* (FF) e balanceamento de carga (LB).

Tabela 3.1: Visão Geral do Estado da Arte

Trabalhos	**Cenário**	**Energia**	**Tempo**	Recursos Esp.	Contadores	DVFS	*Screening*	FF	LB
Molka et al. (2017)		≈4%	≈0,2%	-	X	X	-	-	-
Millani e Schnorr (2016)	Memória	≈9%	-	-	-	X	X	-	-
Wang et al. (2015)	Compartilhada	≈9%	≈1%	X	-	X	X	X	-
Laurenzano et al. (2011)		≈11%	-	-	X	X	-	X	-
Shafik et al. (2015)		≈17%	-	X	X	X	-	-	-
Padoin et al. (2017)		≈19%	-	-	-	X	-	-	-
Etinski et al. (2009)	Memória	≈60%	-	X	-	X	-	-	X
Huang e Feng (2009)	Distribuída	≈11%	≈5	-	-	-	-	-	X
Bhalachandra et al. (2017)		≈31%	-	X	-	-	-	-	X
Li et al. (2013)	Híbridas	≈9	-	X	X	X	-	-	-
Lim, Freeh e Lowenthal (2011)		≈11%	≈2%	-	-	-	-	-	-

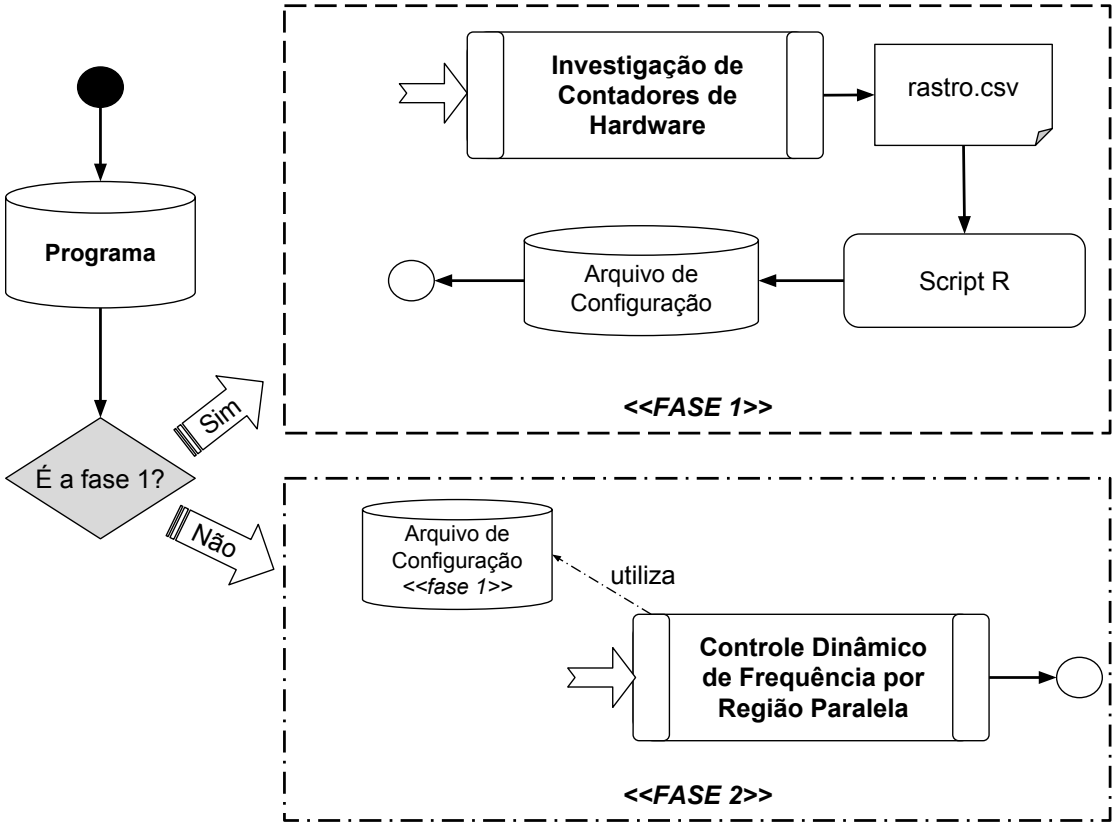
Os valores apresentados pela Tabela 3.1 apresentam uma estimativa geral de quanto os autores conseguiram obter de ganho em energia por técnica utilizada. Vale salientar que, a maior parte das abordagens que reduzem o consumo de energia acarretam sobrecargas no tempo de execução. Na maioria dos casos é possível observar que, os trabalhos

que utilizam técnicas fornecidas pelo próprio hardware resultam em maiores ganhos de energia. Além do que, se limitam a plataformas específicas ou processadores recentes. Nesse contexto, o presente trabalho propõe uma abordagem em formato de *Workflow* que pode ser aplicada em qualquer plataforma que possua suporte a política *Userspace* do *driver* CPUFreq do ambiente Linux.

4 WORKFLOW DE ANÁLISE DE CARACTERÍSTICAS COMPUTACIONAIS

Nesse capítulo aborda o *workflow* construído para identificar as regiões *Memory-Bound* de aplicações paralelas (visão geral do *Workflow* na Figura 4.1). O *Workflow* é dividido em duas etapas, na primeira busca-se a assinatura computacional das regiões paralelas, ou seja, a partir de uma análise por contadores de hardware, o comportamento das regiões paralelas do programa alvo é conhecido. Já na fase 2 utilizamos essa base de conhecimento (obtida na fase anterior) para aplicar um controle dinâmico de frequência de acordo com o comportamento de cada região paralela.

Figura 4.1: Visão Geral do *Workflow*



4.1 Fase 1: Obtenção da Assinatura Computacional

Uma das formas de compreender o comportamento de aplicações paralelas é realizando uma investigação de contadores de hardware, os quais fornecem estimativas de como a aplicação se comporta ao executar em uma plataforma de hardware específica. A partir desses contadores é possível obter medidas como o total de ciclos de CPU, acessos a *Cache* ou memória principal, quantidade de instruções de tipo *load*, entre outras. Na primeira fase do *Workflow*, busca-se investigar o comportamento da aplicação paralela, utilizando para isso contadores de hardware, os quais podem ser escolhidos de acordo com o objetivo da investigação. Como em nossa abordagem desejamos conhecer quais são as regiões da aplicação que são limitadas pela memória, ou seja, as regiões *Memory-Bound*, utilizamos contadores hardware para obter o total de instruções executadas, total de ciclos de CPU, total de acessos a *Cache* L2 e o total de *misses* na *Cache* L2.

De forma geral, ao executar uma região paralela OpenMP, o processo principal dispara várias *threads* de processamento. Em nossa abordagem precisamos conhecer o comportamento das regiões executadas por cada *thread*, desprezando a parte sequencial do programa. Em uma execução preliminar (fase 1), nós realizamos um monitoramento de contadores de hardware, os quais são obtidos por *thread*, a fim de que no final dessa primeira execução, agrupando todos os contadores por *thread*, seja possível identificar as regiões mais limitadas pela memória.

Na maior parte dos casos, a limitação por memória ocorre quando o programa (ou *thread* nesse caso) não consegue encontrar a informação desejada no primeiro nível de *Cache*, com isso o segundo nível é acessado. Caso a informação não esteja lá, será necessário acessar a memória principal, o que gera um custo maior de desempenho, limitando o tempo de execução ao tempo de acesso a memória. Além de investigar a limitação pela memória, a limitação pela CPU ocorre quando é necessário processar um número alto de instruções em um único ciclo de processador, fazendo com que o tempo de execução esteja totalmente relacionado ao tempo de espera pela CPU. A seguir são apresentadas as equações utilizadas para calcular o IPC e a taxa de *misses* na L2.

$$IPC = \frac{Instruction}{Ciclos} \tag{4.1}$$

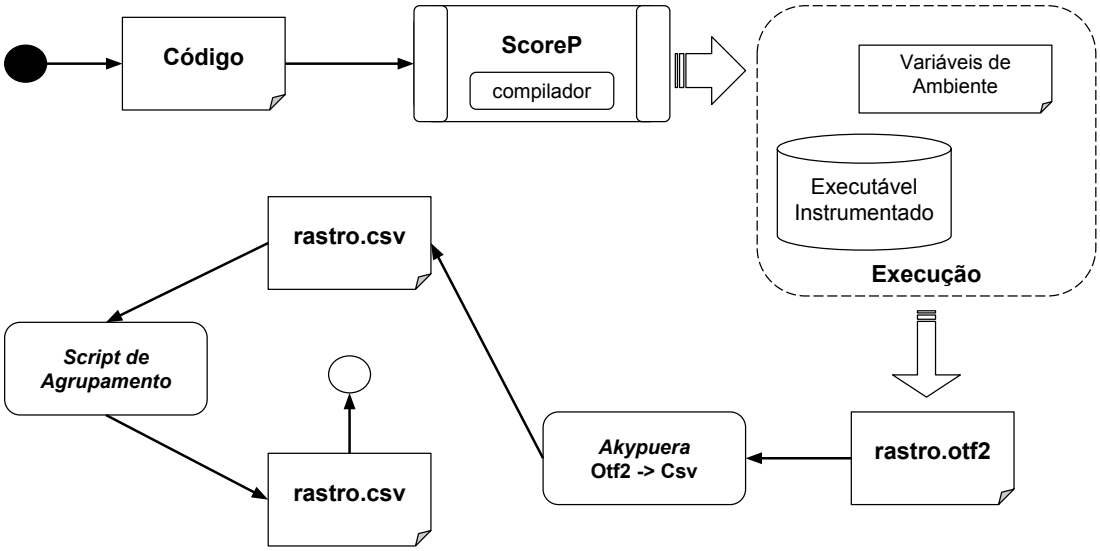
$$MissesRate_{L2} = \frac{Misses_{L2}}{Acesses_{L2}} \tag{4.2}$$

O IPC (*Instruction per Cycle*) permite mapear quantas instruções são realizadas por ciclo de processador, essa medida é extremamente importante para reconhecer em aplicações paralelas, o comportamento *CPU-Bound*. Já a taxa de *misses* permite de maneira geral mapear a porcentagem de insucesso nas requisições realizadas pela aplicação a memória. Em grande parte das plataformas, existem três níveis de memória *Cache*, por essa razão, para definir o comportamento *Memory-Bound* das regiões paralelas, utilizamos essas duas métricas em conjunto. Conforme analisado em Moro e Schnorr (2016), o uso da taxa de *misses* na *Cache* L2 e o IPC permitem um bom mapeamento do comportamento *Memory-Bound* em aplicações paralelas.

A partir da etapa de investigação de contadores é possível obter um rastro que define o comportamento do programa. Esse comportamento consiste no registro dos contadores de hardware para cada região paralela, linha de código e duração temporal da região. Nesse cálculo deve ser considerado as diferentes *threads* da aplicação, as quais geram vários resultados para mesmas regiões de código. Além disso, o cálculo deve garantir que a coleta do contador de hardware ocorra ao entrar e ao sair da região paralela, de forma que seja realizado a subtração do valor obtido na saída do valor inicial, o qual existia ao entrar na região em questão.

A investigação de contadores de hardware pode ser realizada com a ferramenta ScoreP, essa ferramenta permite a coleta de contadores de hardware com PAPI. Nisso, basta informar ao ScoreP quais os contadores a serem utilizados, ao executar a aplicação alvo, o ScoreP coletará os contadores de hardware ao entrar e sair das regiões paralelas para cada *thread*. Com isso, é possível obter um arquivo de rastro em formato OTF2 que descreve os valores de contadores de hardware coletados, por *threads*, por regiões paralelas e linhas de código. Vale salientar que, o ScoreP realiza a instrumentação do arquivo binário, logo é necessário compilar o programa alvo utilizando um "compilador" fornecido pela própria ferramenta. Na Figura 4.2 é possível visualizar a abordagem de investigação de contadores de hardware utilizando a ferramenta ScoreP.

Figura 4.2: Investigação de Contadores de Hardware com ScoreP.

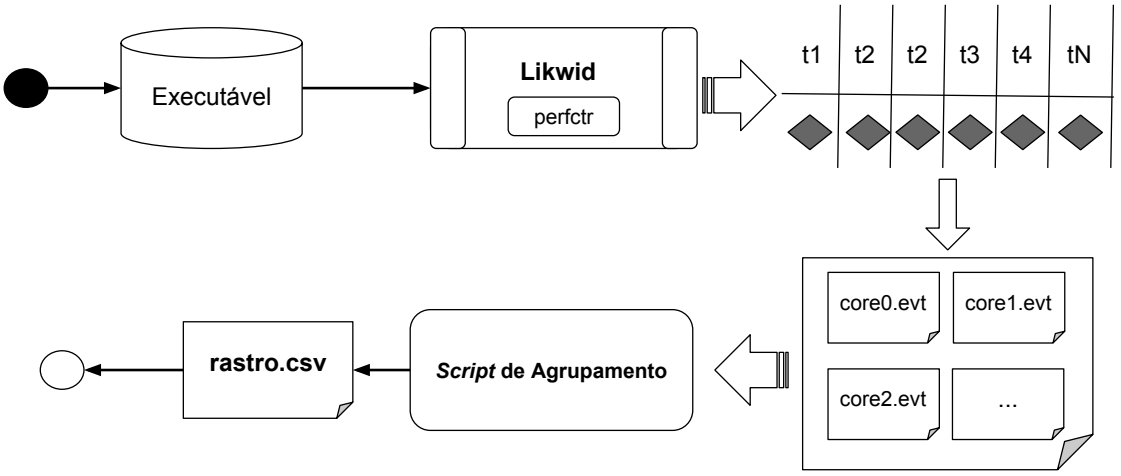


Na Figura 4.2 é possível visualizar que a entrada do fluxograma é o código da aplicação alvo, esse código é compilado utilizando o compilador fornecido pelo próprio ScoreP, dependendo da aplicação, se esta for escrita em C ou em Fortran, existem opções como "scorep-gcc" ou "scorep-gfortran", o arquivo executável gerado pela compilação é um binário instrumentado. Antes da execução desse binário é necessário definir as variáveis de ambiente que serão utilizadas para informar ao ScoreP quais contadores de hardware devem ser investigados, como também o tamanho de memória a ser alocada por processo e demais definições. A partir da execução é gerado um arquivo em formato OTF2, o qual contém o rastro da aplicação, com isso utiliza-se alguma ferramenta para converter esse rastro em formato CSV, em nosso esquema utilizamos a ferramenta Akypuera. Com o rastro CSV é possível identificar a medição de contadores de hardware por *thread* de processamento e por região de código fonte. Caso seja necessário realizar várias execuções para evitar a saturação de contadores de hardware, é necessário utilizar algum *script* de agrupamento para tratar essas informações.

Outra ferramenta interessante ao investigar contadores de hardware é a Likwid, essa ferramenta pode ser utilizada da mesma forma que o ScoreP, a principal diferença é que ao invés de uma instrumentação de binário, a Likwid é uma espécie de ferramenta de execução, sendo necessário especificar o executável do programa a ser analisado. A ferramenta permite a definição do intervalo temporal (milissegundos) da coleta dos contadores de hardware, durante a execução. A partir da ferramenta é possível acessar os eventos Perf que fornecem medidas similares a ferramenta PAPI, sendo possível também analisar taxa de *Cache misses*, quantidade de instruções, ciclos e assim por diante. Na Figura 4.3 é possível visualizar a abordagem de investigação de contadores de hardware utilizando a ferramenta Likwid.

Conforme visto na Figura 4.3, a quantidade de coletas de contadores será realizada de acordo com o tamanho do intervalo temporal definido e com o tempo total de execução da aplicação. A partir disso, a Likwid registra essas amostras para cada *thread*, em arquivos com a extensão "evt". Esses arquivos devem ser analisados com alguma espécie de *script* estatístico adequado para agrupar os diferentes valores de contadores, por amostras de tempo e por *threads* de processamento. Uma das desvantagens da utilização

Figura 4.3: Investigação de Contadores de Hardware com Likwid.



dessa ferramenta é que o mapeamento das regiões paralelas é realizado de acordo com as marcas temporais, não sendo possível correlacioná-las com os respectivos trechos de código-fonte, dificultando a relação dos valores das métricas obtidas durante a execução com o código-fonte. Com isso, a utilização da ferramenta ScoreP é mais vantajosa, visto que ela trata as regiões paralelas e suas respectivas localizações no código fonte.

A fase 1 é finalizada com o arquivo de configuração, o qual é obtido utilizando como entrada o rastro de comportamento. A partir desse rastro será definido quais frequências de processador deverão ser utilizadas para cada região. A partir dos valores dos contadores de hardware, estabelecemos uma relação entre as duas métricas para definir as regiões *Memory-Bound*. Essas regiões recebem uma frequência de processador baixa definida no próprio arquivo de configuração.

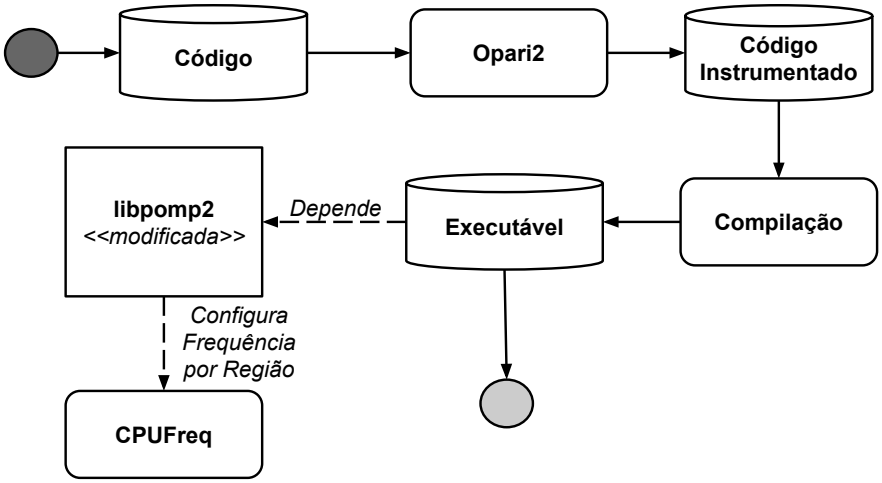
4.2 Fase 2: Controle Dinâmico da Frequência por Região Paralela

O controle dinâmico da frequência é realizado de acordo com o arquivo de configuração gerado na fase anterior. Sendo necessário uma execução preliminar da aplicação para o conhecimento do comportamento das regiões paralelas, a partir disso será possível aplicar a frequência adequada para a respectiva região. As frequências a serem configuradas são armazenadas em uma estrutura de dados, ao entrar em uma região é buscado dessa estrutura de dados a frequência de processador associada a sua respectiva região.

Como a troca de frequência é custosa em termos de desempenho, realizamos ela apenas quando é necessário, levando em consideração a duração da região paralela. Em situações onde a região paralela possui um tempo de execução curto, a frequência não é alterada. A frequência a ser utilizada será a última configurada em uma região anterior. Além disso, a troca de frequência ocorre sempre ao entrar na região paralela, ao sair não é realizado nenhuma configuração de frequência, pois ao entrar em uma nova região paralela, uma nova frequência será atribuída para a aplicação. A Figura 4.4 apresenta uma ilustração do funcionamento do controle dinâmico da frequência em cada região paralela.

Ao executar a ferramenta Opari2 (LORENZ et al., 2014) sobre um determinado código paralelo em OpenMP, o Opari2 gera um código fonte instrumentado, no qual é realizado chamadas a uma biblioteca externa. As chamadas são realizadas ao entrar em

Figura 4.4: Controle Dinâmico de Frequência por Região Paralela.



uma região paralela, barreira de sincronização e outros tipos de regiões. Com suporte de Opari2, essa biblioteca externa pode manter uma estrutura de dados que define as regiões executadas, ela mantém também a linha de código associada ao início daquela região e a linha de finalização da mesma. Dessa forma, qualquer procedimento a ser executado ao entrar em determinada região pode ser codificado nesta biblioteca externa, reescrevendo seu comportamento e potencialmente influenciando a aplicação em tempo de execução. Em nosso cenário, utilizamos o arquivo de configuração gerado anteriormente na fase 1, o qual nos informa as regiões de código fonte e as respectivas frequências de execução, assim basta implementar a biblioteca externa para atribuir, em tempo de execução, para determinada região paralela sua respectiva frequência de processador. Para que isso se torne possível, essa biblioteca externa pode utilizar CPUFreq, a qual fornece o serviço de troca de frequência.

4.3 Conclusão do Capítulo

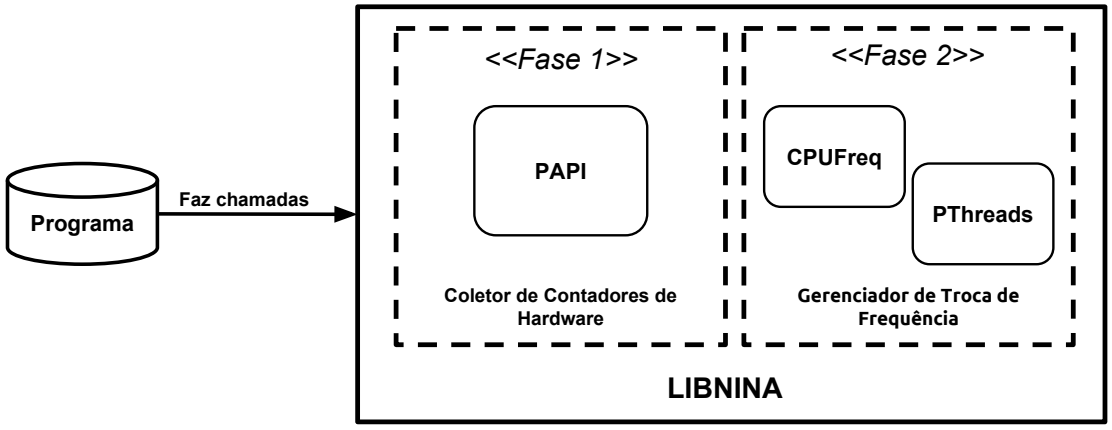
O *Workflow* proposto permite encontrar as regiões limitadas pela memória, com isso, em uma nova execução, essas regiões serão executadas com uma frequência baixa de processador, a fim de obter ganhos em consumo de energia. Para investigar o comportamento *Memory-Bound* em aplicações paralelas é possível utilizar ferramentas como ScoreP ou Likwid. O ScoreP permite o mapeamento da medida de contador de hardware com a respectiva região de código analisada, já o Likwid permite o ajuste temporal em que as coletas serão realizadas, o que possibilita o agrupamento das medidas em diferentes marcas temporais. A principal contribuição do *workflow* está no mapeamento comportamental das regiões paralelas, sendo elas concebidas como faixas temporais (Likwid) ou como trechos de código (ScoreP), o Apêndice A apresenta o uso das duas ferramentas.

Além disso, na fase 2, o *workflow* pode empregar uma biblioteca externa que possibilita a atribuição da frequência de processador exatamente na região desejada, o que permite utilizar o arquivo de configuração gerado na fase anterior, como base de conhecimento. Dessa forma, as regiões identificadas como “potenciais” (*Memory-Bound*) sofrerão a alteração de sua frequência em tempo de execução, para uma frequência baixa, a fim de obter algum ganho no consumo de energia. Essa biblioteca externa é a LibNina, descrita no capítulo seguinte.

5 LIBNINA: DVFS PARA REGIÕES PARALELAS DE APLICAÇÕES OPENMP

Esse capítulo aborda a implementação da biblioteca LibNina, que permite a execução do *workflow* descrito anteriormente (veja Capítulo 4) de maneira totalmente transparente ao programador. Ela se destina exclusivamente para tratar de regiões paralelas de aplicações paralelizadas com OpenMP. A LibNina pode ser utilizada para executar a fase 1 (registro da assinatura computacional das regiões paralelas), como também a fase 2 (controle da frequência do processador) do *Workflow*. Na Figura 5.1 é possível visualizar a arquitetura geral da LibNina, descrita na sequência.

Figura 5.1: Visão Geral da LIBNina.



A Figura 5.1 apresenta o funcionamento geral da LibNina. No início do fluxo, temos o programa que é automaticamente instrumentado com um compilador fonte para fonte (*source-to-source*) através da ferramenta Opari2 (LORENZ et al., 2014). As instrumentações adicionadas automaticamente consistem em chamadas à LibNina em pontos estratégicos, tais como o início e o final de cada região paralela. A cada região paralela é sinalizado quando o fluxo de execução (*thread*) entrou em determinada região e também quando saiu. Além de sinalizar as regiões, a LibNina mantém uma estrutura de dados que representa (em memória) a linha de código de início e fim de cada região paralela. A cada entrada e saída de uma região paralela é disparado uma chamada à LibNina, a fim de que ela possa realizar as operações necessárias de acordo com seus componentes. A biblioteca possui dois componentes principais, o primeiro destinado a investigação da assinatura computacional das regiões paralelas (fase 1), o qual utiliza a biblioteca PAPI para acessar contadores de hardware, já o outro (fase 2) que aplica a cada região paralela uma frequência de processador de acordo com a sua assinatura computacional, utilizando a biblioteca CPUFreq para alterar a frequência de processador.

5.1 Obtenção da Assinatura Computacional das Regiões Paralelas

Para mapear em tempo de execução as próximas regiões a serem executadas, utilizamos a ferramenta Opari2 (LORENZ et al., 2014). Essa ferramenta permite a instrumentação de aplicações paralelas (*source-to-source*), a fim de gerar para cada região paralela, chamadas à biblioteca LibNina. Por ser uma biblioteca dinâmica, ela possui métodos chamados em tempo de execução para mapear o início e o fim de uma região paralela, barreiras de sincronização, *tasks* e outras regiões do programa. A listagem de código 5.1

mostra o código original de uma aplicação simples, em estado original. A listagem de código 5.2 mostra o código com a instrumentação Opari2, realizada automaticamente.

Código 5.1 – Programa Olá Mundo em OpenMP.

```
int main (int argc, char *argv[])
{
    int nthreads, tid;
    #pragma omp parallel private(nthreads, tid)
    {
        tid = omp_get_thread_num();
        printf("Hello_World_from_thread=%d\n", tid);
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number_of_threads=%d\n", nthreads);
        }
    }
}
```

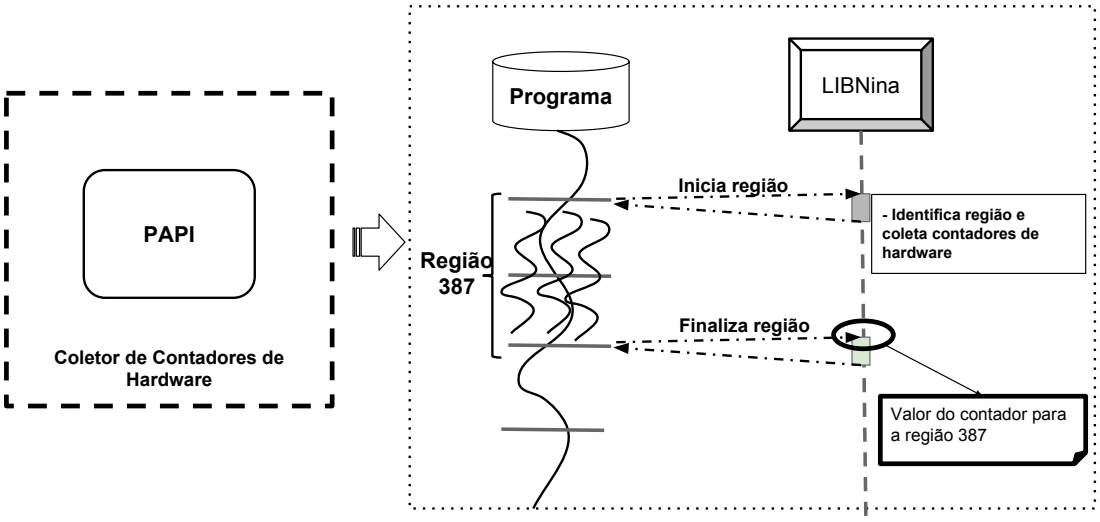
Código 5.2 – Programa Olá Mundo em OpenMP com Chamadas a POMP2.

```
int main (int argc, char *argv[])
{
    int nthreads, tid;
    {
    int pomp2_num_threads = omp_get_max_threads();
    int pomp2_if=1;
    POMP2_Task_handle pomp2_old_task;
    POMP2_Parallel_fork(&pomp2_region_1, pomp2_if, ...)
   #line 10 "path/omp_hello.c"
    #pragma omp parallel private(nthreads,tid) POMP2_DLIST_00001
        ...
    {   POMP2_Parallel_begin(&pomp2_region_1);
        #line 11 "/home/gbmoro/Documents/openmpexample/omp_hello
           .c"
        {
            tid = omp_get_thread_num();
            printf("Hello_World_From_Thread=%d\n", tid);
            if (tid == 0) {
                nthreads = omp_get_num_threads();
                printf("Number_Of_Threads=%d\n", nthreads);
            }
        }
        { POMP2_Task_handle pomp2_old_task;
        POMP2_Implicit_barrier_enter(&pomp2_region_1,...);
        #pragma omp barrier
        POMP2_Implicit_barrier_exit(&pomp2_region_1,... ); }
        POMP2_Parallel_end(&pomp2_region_1); }
        POMP2_Parallel_join(&pomp2_region_1, pomp2_old_task);
    }
        #line 24 "path/omp_hello.c"
}
```

O código instrumentado com Opari2 (veja listagem 5.2) apresenta a instrumentação através de chamadas `POMP2` para cada região paralela. A partir dessas chamadas a LibNina é acionada, permitindo a execução das fase 1 ou 2 de nossa biblioteca. Chamadas como "POMP2 Implicit barrier enter", "POMP2 Implicit barrier exit", "POMP2 Parallel join" são alguns exemplos das chamadas POMP2 que contatam a LibNina. Além dessas chamadas, visualizamos também alguns parâmetros encaminhados nessas chamadas, como por exemplo o "pomp2_region_1", o qual define a estrutura de dados que identifica aquela região paralela. Como a LibNina é uma biblioteca dinâmica, ao executar a aplicação indicamos sua localização para que ela possa "atuar sobre" a execução do programa instrumentado com Opari2.

Ao habilitar a fase 1, com os contadores de hardware pré-definidos via arquivo de configuração, a LibNina realizará a coleta desses contadores para cada *thread* da aplicação, ao entrar em uma região paralela (*Parallel fork*) e ao sair da mesma (*Parallel Join*). No final da região paralela será gerado um rastro que descreverá a linha de código inicial da região, o tempo final, o tempo inicial e também a duração da região paralela, como também os valores dos contadores de hardware coletados. Com essas informações é possível estimar o comportamento da região (de acordo com os contadores de hardware utilizados), como também a duração da região paralela. Essas informações são fundamentais, pois ao realizar a troca de frequência (fase 2) poderá ser gerado uma sobrecarga significativa para regiões *Memory-Bound* de curta duração. A Figura 5.2 apresenta uma ilustração do funcionamento do módulo coletor de contadores de hardware.

Figura 5.2: Módulo Coletor de Contadores de Hardware - LIBNina.



Conforme é possível visualizar na Figura 5.2, a coleta de contadores de hardware é realizada por *thread* de processamento, ao início e ao fim da região paralela. Com os valores obtidos por *thread* de processamento, realizamos a soma desses valores, a fim de obtermos um valor representativo para toda a região 387 (no exemplo da Figura).

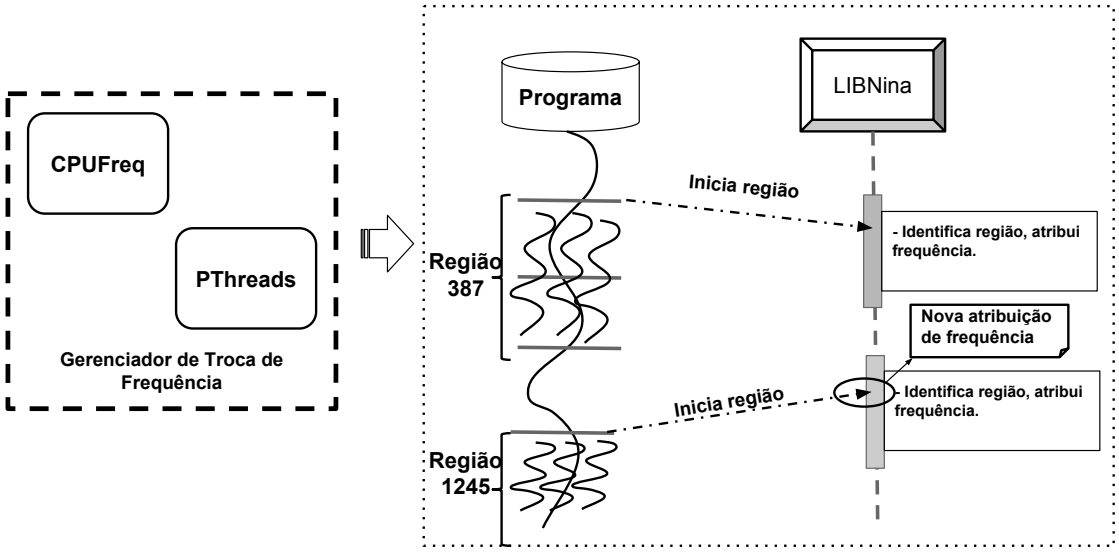
5.2 Implementação das Frequências Adequadas por Região

A fase 2 é o modo de execução em que a LibNina realiza a troca de frequência nas regiões *Memory-Bound*. O controle dinâmico de frequência é realizado em tempo de execução, ao entrar em cada região paralela é verificado na estrutura de dados, se essa

região (composta por arquivo e linha de código) deverá sofrer uma troca de frequência. Caso sim, na mesma estrutura de dados será recuperada a frequência que será atribuída à mesma.

A troca de frequência é realizada utilizando a biblioteca CPUFreq, a qual permite a troca de frequência utilizando a política *Userspace*. Para realizar esse tipo de operação, o programa deve ser executado com a LibNina, como usuário administrador, pois essa operação necessita de permissões no Sistema Operacional Linux. Vale salientar que, os processadores que sofrerão a troca de frequência devem ser especificados na LibNina via variável de ambiente, para que seja possível indicar o "identificador" das CPUs que sofrerão a alteração de frequência durante a execução do programa. A Figura 5.3 apresenta um esquema do funcionamento do módulo gerenciador de troca de frequência.

Figura 5.3: Módulo Gerenciador de Troca de Frequência - LIBNina.



O controle dinâmico de frequência é realizado por uma *thread* específica, visto que essa operação acarreta uma sobrecarga significativa em tempo de execução, como também em consumo de energia. Para amenizar essa situação, implementamos um serviço assíncrono que executa paralelo a aplicação, esperando que a LibNina sinalize os momentos exatos para efetuar a troca de frequência. A troca de frequência é realizada apenas ao entrar nas regiões paralelas (Figura 5.3), dessa maneira evitamos a troca de frequência "extra" no final da região paralela, pois sempre quando termina uma região paralela outra inicia. Vale salientar que, a LibNina não garante que quando a região paralela começar, a nova frequência já estará atualizada, pois existe um tempo de atraso entre a requisição pela troca de frequência e a sua aplicação nos processadores.

5.3 Conclusão do Capítulo

A LibNina é a implementação do *Workflow* proposto nessa dissertação, ela atua como uma biblioteca dinâmica, possibilitando o seu uso em qualquer aplicação OpenMP. Além disso, dividimos o funcionamento da biblioteca em dois componentes, o primeiro que atua na fase de obtenção da assinatura computacional da aplicação (fase 1 do *Workflow*) e o outro que realiza o controle dinâmico de frequência (fase 2). Os dois componentes da biblioteca são independentes, sendo executados individualmente (mediante controle

através de variáveis de ambiente). O componente de obtenção da assinatura computacional analisa contadores de hardware, os quais podem ser configurados de acordo com o tipo de análise que se deseja realizar. Já o componente de troca de frequência atua de forma específica em determinadas regiões paralelas. Vale salientar que, pelo fato de existir uma sobrecarga ao trocar a frequência do processador, esse componente troca a frequência em uma *thread* auxiliar, de maneira assíncrona, com o objetivo de amenizar tal sobrecarga.

Além disso, a biblioteca só pode ser utilizada com aplicações OpenMP, instrumentadas com a ferramenta Opari2. Em contrapartida a este ponto, a LibNina pode entrar em contato com qualquer região de uma aplicação OpenMP, sendo essa uma barreira de sincronização, tarefa, entre outras. Logo, essa característica permite que outras abordagens possam ser incorporadas a LibNina, tanto em termos de caracterização de aplicações, como também de técnicas para redução do consumo de energia. Como exemplo de uso dessa característica de instrumentação por regiões, as regiões de barreiras de sincronização entre *threads* que poderiam ser uma oportunidade para reduzir o consumo de energia em aplicações com uma alta taxa de dependência entre os fluxos de execução.

6 VALIDAÇÃO E RESULTADOS EXPERIMENTAIS

Esse capítulo apresenta a utilização da LibNina com a aplicação Lulesh, que possui um número considerável de regiões paralelas não aninhadas. Esse cenário permite listar os resultados obtidos a partir da exposição da LibNina a várias configurações de troca de frequência por região paralela. Apresenta-se também os principais "gargalos" ao utilizar a abordagem proposta, deixando claro seus limites.

A plataforma utilizada para executar os experimentos é uma *Workstation* de 126 GBytes de memória RAM, dois processadores Intel(R) Xeon(R) CPU E5-2650v3 organizados em NUMA (*Non-Uniform Memory Access*), cada um com 10 núcleos físicos com a tecnologia *HyperThreading* (HT). A máquina permite a frequência máxima de 2.30GHz e mínima de 1.2GHz. Além disso, a plataforma possui três níveis de *Cache*: L1 de instruções e L1 de dados, ambas com 32KB; L2 com 256KB e L3 com 25MB. Cada núcleo físico da plataforma possui internamente uma *Cache* L1 de instruções e dados, próxima a ela está a *Cache* L2 e a *Cache* L3 é compartilhada com os demais recursos do processador. Vale salientar que, cada um dos processadores possui memória RAM de 63GB.

Quanto as características de software, a plataforma possui Sistema Operational Ubuntu 16.04LTS com *kernel* Linux 4.4.0; compilador GNU *Compiler Collection* - GCC 5.4.1 com OpenMP; *framework* CPUFreq com suporte as políticas *Ondemand*, *Conservative*, *Performance* e *Userspace*, este último utilizado para alterar a frequência dos processadores de maneira explícita; a ferramenta *AppPowerMeter* para medir o consumo de energia (que utiliza os contadores disponíveis através da interface Intel RAPL); e *Performance Application Programming Interface* - PAPI 5.4.3 para coletar os contadores de hardware.

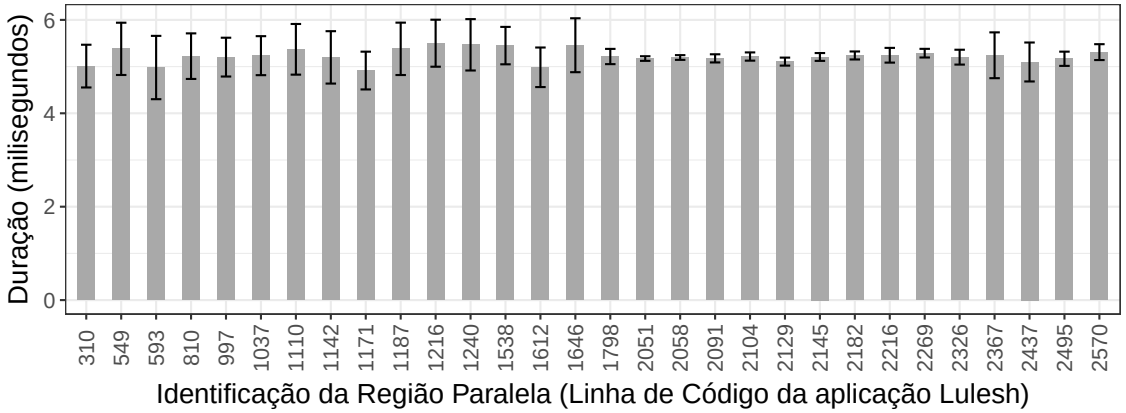
6.1 Conhecendo a Aplicação Lulesh

A aplicação utilizada nos experimentos é o *benchmark* Lulesh (KARLIN; KEASLER; NEELY, 2013), o qual representa uma simulação computacional de modelagem hidrodinâmica. Essa simulação reproduz uma onda de explosão em um espaço confinado, utilizando como base o método de Sedov (LLNL, 2018). A aplicação é reconhecida por possuir aspectos *Memory-Bound*. Por essa razão, ela foi utilizada pela perspectiva de ganhos potenciais, ao executar tais regiões em uma frequência de processador mais baixa. Vale salientar que, a aplicação possui um total de 30 regiões paralelas, sendo que nenhuma dessas regiões possui outras regiões aninhadas.

O gráfico da Figura 6.1 apresenta um demonstrativo da duração média em milissegundos (no eixo vertical) de cada uma das regiões de código (no eixo horizontal, identificadas pelo número da linha de código) para uma execução com tamanho de entrada 10, utilizando 20 *threads*. No gráfico é possível identificar que as regiões que levam mais tempo são as que ocorrem nas linhas 1216, 1240, 1538 e 1798. Já as menores regiões ocorrem nas linhas 1171, 593 e 1612. De forma geral, analisando uma execução que durou 9 minutos, as regiões apresentam uma duração similar, pois elas variam na faixa de 4.9 a 5.5 milissegundos (intervalo pequeno).

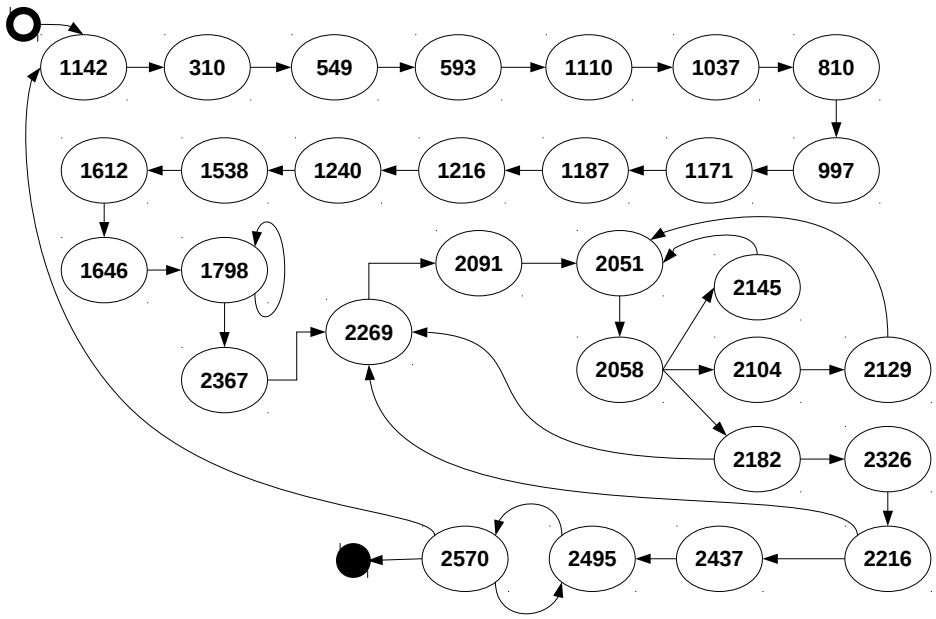
Como as regiões paralelas da aplicação são executadas várias vezes em determinada ordem, é possível estabelecer um grafo dessas regiões, conforme pode ser observado na Figura 6.2. O grafo apresenta a ordem em que as regiões são executadas pela aplicação Lulesh, os nós são nomeados de acordo com a respectiva linha de código da região paralela. O fluxo estende-se sequencialmente da região 1142 até a região 1798, onde ocorre

Figura 6.1: Duração das Regiões Paralelas - Lulesh.



algumas iterações que podem estar relacionadas com o tamanho de entrada da aplicação. Após essa região, a partir da região 2058 são definidos três nós, esses três "caminhos" podem resultar em mais iterações. O último nó executado é a região 2570 que pode levar ao nó raiz, reinicializando o fluxo da aplicação.

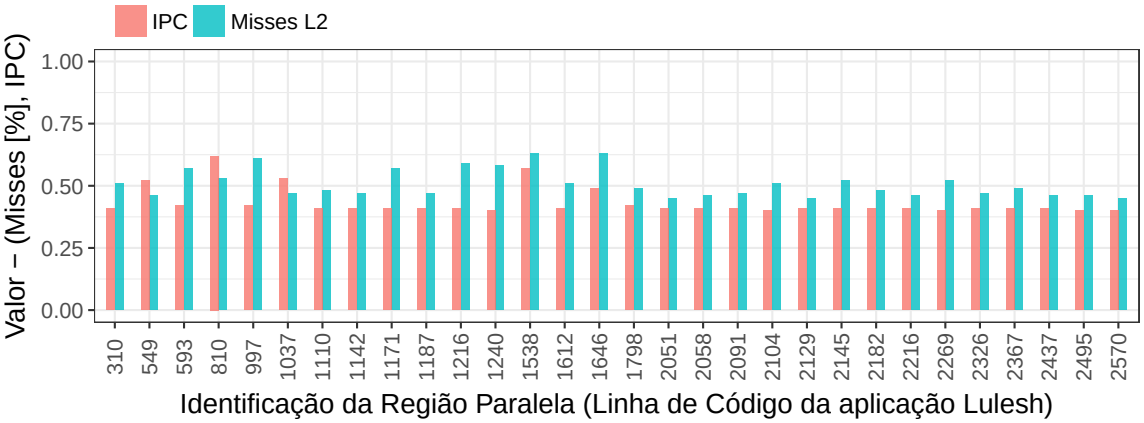
Figura 6.2: Grafo de Regiões Paralelas - Lulesh.



A escolha da frequência de processador mais adequada para determinada região paralela deve levar em consideração sua duração, a ordem em que as regiões são conectadas, como também o comportamento *Memory-Bound* definido pelas métricas de hardware. Na Figura 6.3 é possível visualizar a relação entre as duas medidas obtidas: o IPC e a taxa de *Misses* na *Cache* L2. A partir dessa investigação, adotamos várias estratégias diferentes para economia de energia. Cada estratégia se reflete em um arquivo de configuração diferente para a segunda fase de nosso *Workflow*. Dentre as estratégias, as mais proeminentes são: `(a)` o uso de frequência máxima nas regiões onde o IPC é maior do que 0,5, sendo que nas demais localizações o uso de frequência mínima; `(b)` o uso

de frequência máxima em todas localizações exceto em regiões onde a taxa de *misses* é maior do que 50% (nessas regiões o uso de frequência mínima); `(c)` e a combinação das duas configurações anteriores, uso de frequência baixa quando o IPC é menor do que 0,5 e a taxa de misses é maior do que 50%, nas demais regiões o uso de frequência máxima.

Figura 6.3: IPC e Taxa de Misses na Cache L2 por Linha de Código.



6.2 Resultados Experimentais

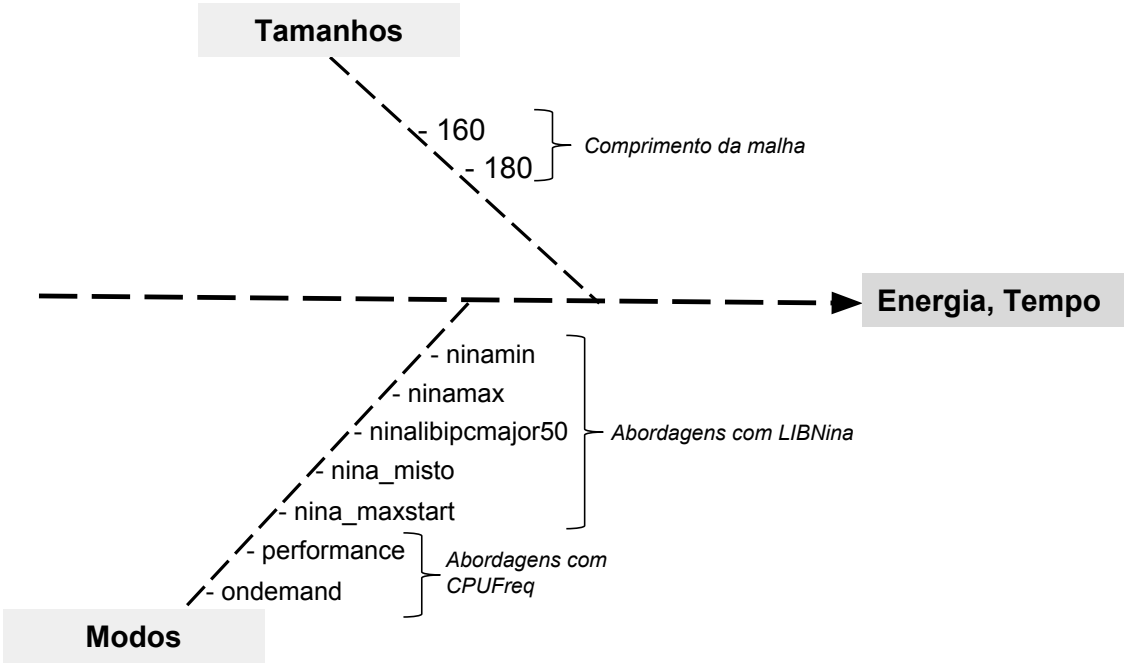
Nessa seção são apresentados os principais experimentos realizados, os quais possibilitam a validação das funcionalidades do *Workflow* proposto e da LibNina (sua implementação). A segunda fase do *Workflow* tem por objetivo realizar o controle dinâmico de frequência, de acordo com o comportamento das regiões paralelas, utilizando duas variações de frequência de processador (baixa ou alta). O primeiro experimento tem por objetivo comparar as diferentes configurações da LibNina com as principais políticas implementadas pelo CPUFreq. Já no segundo experimento realizamos uma comparação da melhor configuração da LibNina com a política *Ondemand*, fornecida pelo CPUFreq. Essa comparação é fundamental, pois a política *Ondemand* é amplamente utilizada em aplicações paralelas, ela propõe um ajuste de frequência de processador de acordo com sua taxa de uso.

6.2.1 Projeto Experimental

Para mapear todas as combinações de configurações a serem executadas, nós mapeamos em um projeto experimental (*Design of Experiments*), os fatores a serem testados no experimento. Com isso, utilizando o pacote `DoE.base` da linguagem R foi possível gerar um arquivo modelo para orientar nosso *script* de execução. Cada linha desse arquivo, define uma combinação de fatores de execução que podem se repetir ao decorrer do experimento, de acordo com o número de execuções definido e a aleatoriedade das repetições. O projeto experimental pode ser visualizado na Figura 6.4 apresentando um diagrama que descreve as configurações utilizadas: dois fatores (Tamanhos e Modos) e seus respectivos valores. Os valores medidos em saída são a energia consumida e o tempo de execução.

Todas as combinações de configuração para os experimentos foram executadas utilizando 20 *threads* fixadas nos núcleos de processamento, com a ausência de *HyperTh-*

Figura 6.4: Diagrama de Causa e Efeito - Projeto Experimental.



reading, que é desativado explicitamente para diminuir a variabilidade experimental. Para tratar a variabilidade natural do sistema computacional, ambos experimentos foram replicados e médias aritméticas simples são calculadas e relatadas. A escolha do número de *threads* está relacionada com a plataforma, a qual possui 20 núcleos físicos de processamento. Os modos de configuração utilizados são definidos da seguinte maneira:

- `ninamin`: todas as regiões são executadas em baixa frequência (com LibNina);
- `ninamax`: todas as regiões são executadas em alta frequência (com LibNina);
- `ninalibipcmajor50`: regiões com IPC a cima de 0,5 possuem alta frequência, as demais possuem baixa frequência (com LibNina);
- `nina_misto`: regiões com IPC a cima de 0,5 possuem alta frequência, as regiões com a taxa de *misses* maior que 50% possuem baixa frequência, em ambas condições a região é executada em baixa frequência;
- `ondemand`: uso da política *Ondemand* do CPUFreq (sem LibNina);
- `nina_maxstart`: uso de alta frequência em regiões específicas do início da execução, as demais são executadas em baixa frequência (com LibNina).

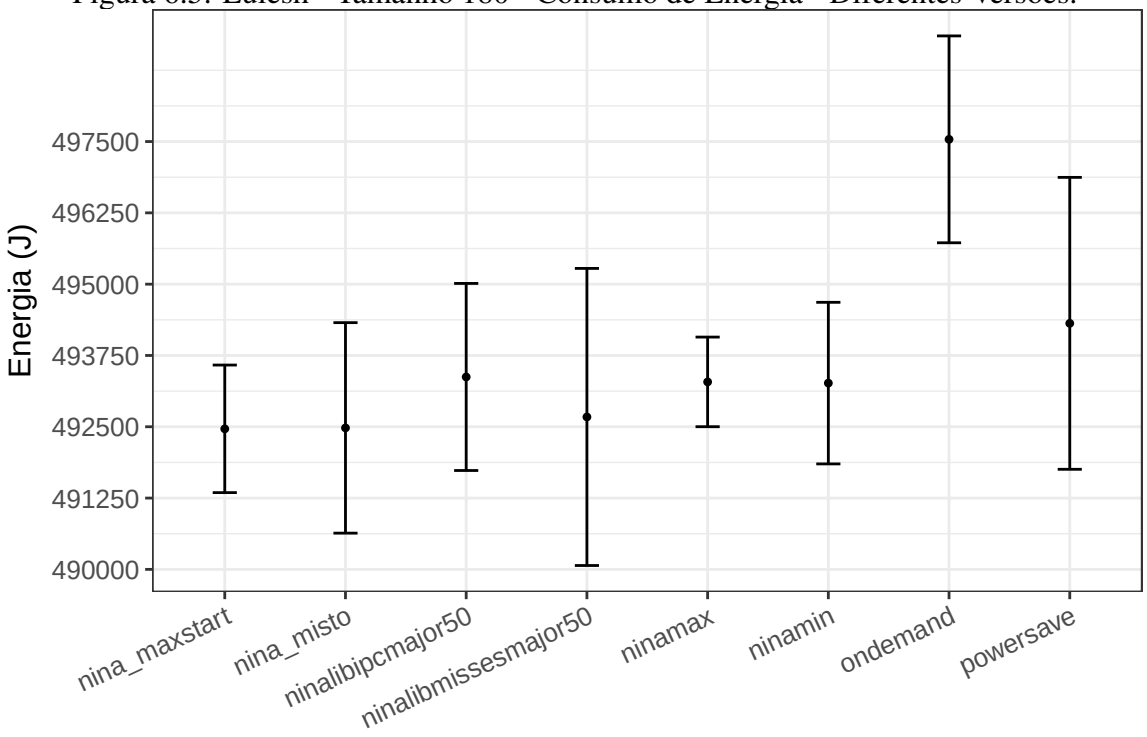
6.2.2 Comparação das Diferentes Configurações da LibNina com as Políticas de Troca de Frequência

Nas execuções é utilizado o tamanho 180, no qual é possível obter execuções que duram aproximadamente 2 horas, utilizando 20 *threads*. Ao total o projeto experimental define 54 execuções, executadas aleatoriamente, a fim de evitar que influências externas ao experimento afetem uma única configuração. Como cada execução possui uma longa duração, definimos um total de 3 replicações para cada configuração. O experimento tem por objetivo comparar diferentes configurações de arquivos para a LibNina com as

políticas *Ondemand* e *Powersave*.

Na Figura 6.5 é possível visualizar os resultados de consumo de energia (no eixo vertical) para diferentes configurações da LibNina e as políticas *Powersave* e *Ondemand* (no eixo horizontal). Os pontos no gráfico representam a média do consumo energético dentre as execuções, e as barras de erro são configuradas assumindo um intervalo de confiança (CI) de 99%. Dentre as duas políticas implementadas no sistema operacional, a política *Powersave* apresenta a melhor opção em termos de consumo energético. No comparativo com as alternativas oferecidas pela LibNina, as melhores são a *nina_maxstart* e a *nina_misto*, a pequena diferença entre elas não é perceptível no gráfico. Logo, a melhor alternativa em termos de consumo de energia com a LibNina, comparado ao uso da política *Ondemand* e as demais configurações é a utilização da configuração *nina_maxstart*.

Figura 6.5: Lulesh - Tamanho 180 - Consumo de Energia - Diferentes Versões.

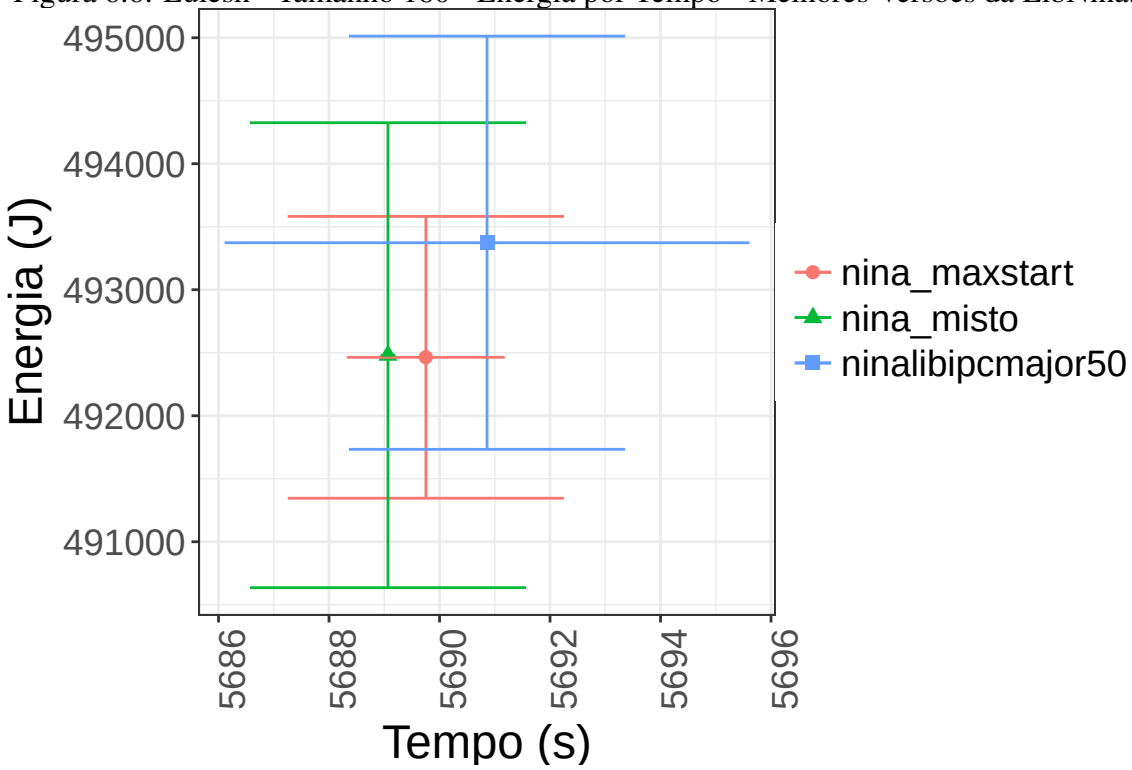


Outro fator interessante é que a política *Powersave* não apresentou o melhor resultado em consumo de energia, como o tempo de execução é proporcional ao consumo de energia e a política utiliza apenas baixa frequência em todas regiões da aplicação, em alguns casos a escolha pela baixa frequência poderia ter sido motivo de perdas em energia. Em alguns casos, como o cenário de regiões mais limitadas pela CPU (*CPU-Bound)*, a escolha pela baixa frequência poderia atrasar significativamente a execução de uma região paralela, com isso mais energia seria necessária, logo não seria proveitoso escolher tal configuração de frequência.

A Figura 6.6 apresenta a relação de consumo de energia por tempo de execução para as melhores configurações da LibNina. O gráfico apresenta intersecções das barras no eixo de tempo de execução, o que não permite uma comparação exata em termos estatísticos. Considerando apenas o valor de média para ambos fatores (consumo de energia e tempo de execução), é possível perceber que, a configuração que apresenta o menor tempo de execução é a *nina_misto*, já a que tem maior gasto em tempo de execução é a

ninalibipcmajor50. Quanto ao consumo de energia, a configuração de melhor resultado é a *nina_maxstart* (cerca de 492463 Joules), diferente disso, o consumo de energia da *ninalibipcmajor50* é o maior comparado aos demais. Como o objetivo do trabalho é o consumo de energia, nesse contexto, o melhor cenário é utilizar a LibNina com a configuração *nina_maxstart*.

Figura 6.6: Lulesh - Tamanho 180 - Energia por Tempo - Melhores Versões da LibNina.



Geralmente, quando se reduz o consumo de energia de uma aplicação, o tempo de execução é afetado. No caso da melhor configuração da LibNina (*nina_maxstart*), a sobrecarga no tempo de execução é de 1,07% comparado a versão com *Ondemand*, utilizando a entrada 180. A sobrecarga é inevitável, visto que em nossa abordagem, ao reduzir a frequência em regiões específicas da aplicação, as mesmas são executadas mais lentamente, dessa forma a energia é poupada pela minimização do uso de recursos, o que gera um tempo de execução maior.

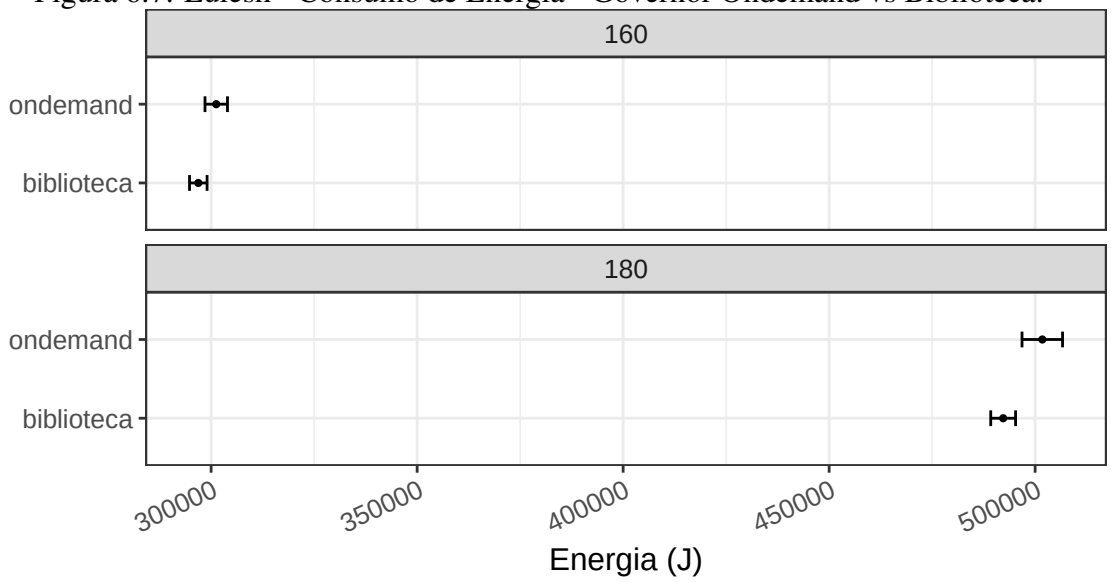
6.2.3 Comparação da LibNina com a Política Ondemand

Nas execuções foram utilizados os tamanhos 160 e 180, pois neles é possível realizar execuções de aproximadamente 2 horas, utilizando 20 *threads*. Ao total o projeto experimental define 54 execuções, as quais são realizadas de forma aleatória, a fim de evitar qualquer influência possível que pode ser causada pela repetição de uma mesma configuração. É definido um total de 5 replicações para cada configuração de tamanho (160 e 180) e modo. O experimento tem por objetivo comparar de forma isolada a melhor configuração da LibNina com a política *Ondemand*.

A Figura 6.7 apresenta um comparativo da política de gerenciamento da frequência do Linux *Ondemand* e da biblioteca proposta, ao utilizar um tamanho de 180 elementos para a aplicação Lulesh. Esse experimento utilizou cinco execuções aleatórias para os

dois modos (*Ondemand* e nossa biblioteca), com duas configurações de entrada (160 e 180 elementos), totalizando vinte execuções. A partir dos resultados é possível verificar que, ao utilizar 160 elementos no Lulesh, o uso da biblioteca reduziu cerca de aproximadamente 1.44% o consumo de energia, comparado com o modo *Ondemand*. Já utilizando 180 elementos, ao utilizar a biblioteca, o consumo de energia reduziu em 1.89%. Embora pequenos, os ganhos são consistentes tendo em vista a longa duração dos experimentos.

Figura 6.7: Lulesh - Consumo de Energia - Governor Ondemand vs Biblioteca.



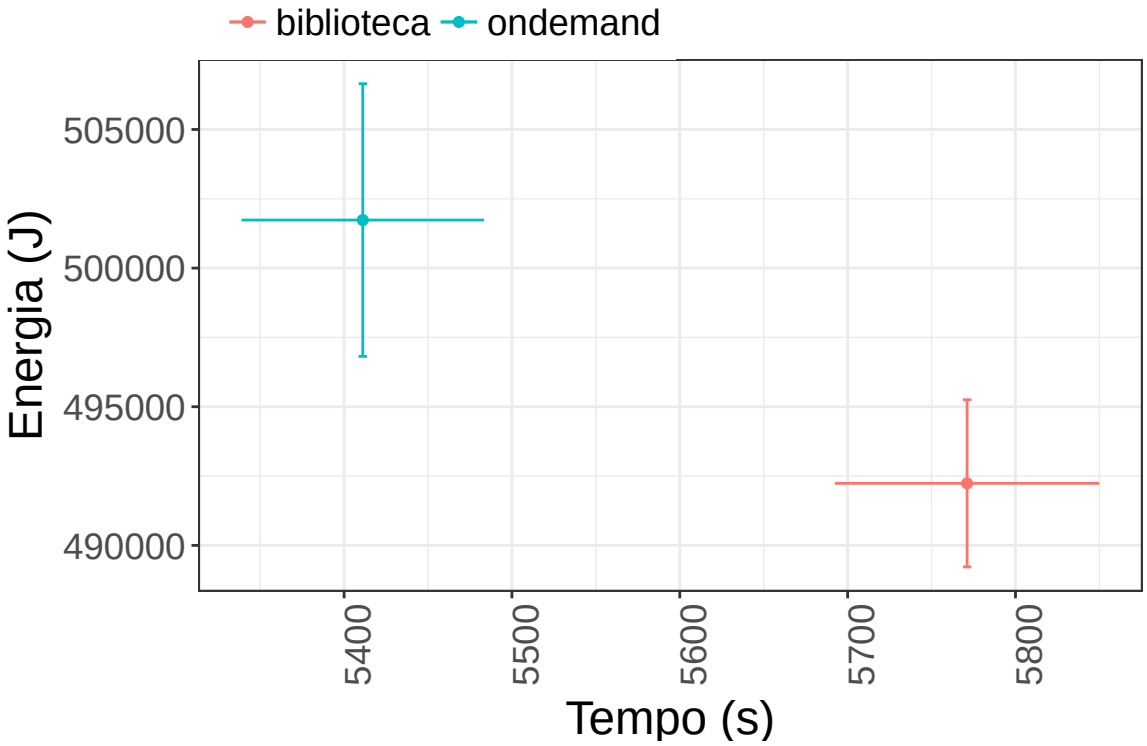
Na Figura 6.8 é possível visualizar um demonstrativo que relaciona o consumo de energia (no eixo vertical) com o tempo de execução (no eixo horizontal) quando adotamos nossa biblioteca e a referência comparativa (*Ondemand*). O uso da biblioteca possibilita um ganho no consumo de energia, conforme visto anteriormente, mas em termos de tempo de execução, o modo *Ondemand* é aproximadamente 0,09% mais rápido, logo a biblioteca apresenta uma sobrecarga de 1,06% no tempo de execução. No gráfico 6.8 é possível visualizar essa relação entre o consumo de energia, como também a perda correspondente no tempo de execução, da biblioteca sobre a política *Ondemand*.

6.3 Conclusão do Capítulo

As diferentes configurações testadas (subseção 6.2.2) possibilitaram identificar uma sobrecarga na troca de frequência, presente na implementação da biblioteca LibNina. A razão desta sobrecarga tem origem na própria aplicação Lulesh, que possui regiões cuja assinatura computacional é bastante diferente (Figura 6.3). Isso obriga a LibNina a adaptar a frequência do processador em mais ocasiões em um intervalo de tempo curto, gerando um custo extra no consumo de energia. Com isso, nós identificamos que em regiões de curta duração (tempo), caso seu comportamento solicitasse uma troca de frequência, seria mais vantajoso executar tal região sem a troca de frequência, a fim de evitarmos o sobrecusto. Os melhores resultados foram obtidos com esta estratégia adaptada que combina a assinatura computacional (contadores de hardware) com a duração das regiões paralelas (*nina_maxstart*).

Ao comparar a LibNina com a política *Ondemand* (subseção 6.2.3), os resultados obtidos com a aplicação Lulesh na comparação preliminar indicam um ganho de 1,89%

Figura 6.8: Lulesh - Tamanho 180 - Tempo e Energia.



no consumo de energia sobre a *Ondemand* do Linux, utilizada como referência. Vale salientar que, ocorreu um aumento no tempo de execução, cerca de 0,09%, no pior caso testado. Acreditamos que esse aumento no tempo de execução esteja relacionado a sobrecarga imposta pela implementação da biblioteca e aos frequentes comandos de adaptação da frequência do processador.

7 CONSIDERAÇÕES FINAIS

Desempenho e consumo de energia são aspectos fundamentais em sistemas de computação. Na maioria das vezes, ao reduzir o consumo de energia, o desempenho é afetado negativamente, como também, em alguns casos ao melhorar o desempenho, o consumo de energia aumenta. Algumas abordagens procuram encontrar o equilíbrio entre esses dois aspectos. Nesse contexto, o presente trabalho buscou como foco principal otimizar o consumo energia em aplicações paralelas, de forma que, o custo no tempo de execução fosse o menor possível.

O principal objetivo desse trabalho foi projetar e implementar um *Workflow* de investigação de regiões paralelas em aplicações OpenMP. Organizada em duas fases, a partir dos contadores de hardware (fase 1), a metodologia propõem a alteração da frequência de execução das regiões (fase 2) para uma configuração adequada a sua assinatura computacional: caso a região paralela seja limitada pela memória, reduz-se a frequência de execução do processador naquele trecho. A abordagem foi implementada na biblioteca LibNina, de código aberto, a qual utiliza informações a respeito do comportamento da aplicação fornecida pela primeira fase e depois altera a frequência de processador do programa de acordo com o comportamento respectivo de cada região paralela, alterando entre frequência máxima e mínima.

Os resultados obtidos com a aplicação Lulesh indicam um ganho de 1,89% no consumo de energia sobre a política *Ondemand* do Linux, utilizada como referência. Vale salientar que, ocorreu um aumento no tempo de execução, cerca de 0,09%, no pior caso testado. Acreditamos que esse aumento no tempo de execução esteja relacionado a sobrecarga imposta pela implementação da biblioteca e aos frequentes comandos de adaptação da frequência do processador.

7.1 Contribuições

A principal contribuição deste trabalho foi uma abordagem em formato de *Workflow*, a fim de que seja possível por ela, a investigação comportamental de uma aplicação paralela em uma granularidade de regiões de código. O comportamento da aplicação é investigado utilizando contadores de hardware, os quais podem ser escolhidos de forma livre, de acordo com o propósito da investigação. Nós utilizamos contadores que possibilitaram o mapeamento das regiões *Memory-Bound* da aplicação, mas seria possível por exemplo, escolher contadores de hardware que permitissem o mapeamento de regiões *CPU-Bound*. Uma biblioteca de código aberto[1] chamada `LibNina` foi desenvolvida para implementar o *Workflow* proposto.

- Produção Bibliográfica:
  - **Moro, Gabriel** e Schnorr, Lucas. Análise de Características Comportamentais de Aplicações OpenMP para Redução do Consumo de Energia. *WPerformance*, 2018.
  - Silveira, Dieison; Bampi, Sergio; **Moro, Gabriel**; Cruz, Eduardo H. M.; Navaux, Philippe O. A.; Schnorr, Lucas M. *IEEE International Conference on Eletronics, Circuits and Systems (ICECS)*, 2016.
  - Silveira, Dieison; **Moro, Gabriel**; Cruz, Eduardo H. M.; Bampi, Sergio; Na-

[1] Código disponível em <https://github.com/tido4410/libnina>

vaux, Philippe O. A.; Schnorr, Lucas M.. Energy Consumption Estimation in Parallel Applications: an Analysis in Real and Theoretical Models. *Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)*, 2016.

- **Moro, Gabriel** e Schnorr, Lucas. Detecting Memory-Bound Parallel Regions to Improve the Energy-Efficiency of Applications. *XV Workshop de Processamento Paralelo e Distribuído (WSPPD)*, 2017.
- **Moro, Gabriel** e Schnorr, Lucas. Measuring Hardware Counters for HPC Application Phase Detection. *XIV Workshop de Processamento Paralelo e Distribuído, (WSPPD)*, 2016.
- Krause, Arthur; **Moro, Gabriel**; Schnorr, Lucas. Análise do Consumo Energético de Aplicações Paralelas com Diferentes Versões de Compiladores. *Escola Regional de Alto Desempenho (ERAD)*, 2017.
- Krause, Arthur; **Moro, Gabriel**; Schnorr, Lucas. Análise de Desempenho da Multiplicação de Matrizes por Strassen contra o Método Tradicional. *Workshop de Processamento Paralelo e Distribuído, (WSPPD)*, 2016.

7.2 Trabalhos Futuros

Como trabalhos futuros, pretendemos investigar os seguintes pontos:

- Alternativas para reduzir o custo da troca de frequência em regiões paralelas de duração menor ou igual ao tempo gasto pela troca de frequência;
- Uso de outros contadores de hardware, como por exemplo, a quantidade de *loads* e *stores* realizados pela aplicação em cada região paralela;
- Uso de OMPT (EICHENBERGER; MELLOR-CRUMMEY; SHULZ, 2014) para remover a dependência Opari2 da `LibNina`, assim o código-fonte da aplicação alvo seria preservado, não sendo necessário o uso de um instrumentador para mapear o início e o fim das regiões paralelas.

REFERÊNCIAS

ADDISON, C. et al. Openmp 3.0 tasking implementation in openuh. In: **Open64 Workshop at CGO**. [S.l.: s.n.], 2009. v. 2009.

BHALACHANDRA, S. et al. An adaptive core-specific runtime for energy efficiency. In: IEEE. **Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International**. [S.l.], 2017. p. 947–956.

BRODOWSKI, D. 2013. Disponível em: <https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt>.

CHANDRA, R. **Parallel Programming in OpenMP**. Morgan Kaufmann Publishers, 2001. (High performance computing). ISBN 9781558606715. Disponível em: <https://books.google.com.br/books?id=vuAY5C5C1W0C>.

DIAMOND, J. et al. Evaluation and optimization of multicore performance bottlenecks in supercomputing applications. In: IEEE. **Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on**. [S.l.], 2011. p. 32–43.

EICHENBERGER, A.; MELLOR-CRUMMEY, J.; SHULZ, M. 2014. Disponível em: <https://www.openmp.org/wp-content/uploads/ompt-tr2.pdf>.

ETINSKI, M. et al. Power-aware load balancing of large scale mpi applications. In: IEEE. **Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on**. [S.l.], 2009. p. 1–8.

HUANG, S.; FENG, W. Energy-efficient cluster computing via accurate workload characterization. In: IEEE COMPUTER SOCIETY. **Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid**. [S.l.], 2009. p. 68–75.

HUTCHESON, A.; NATOLI, V. Memory bound vs. compute bound: A quantitative study of cache and memory bandwidth in high performance applications. 2011.

JANSEN, T. **Basic Parallel Programming with OpenMP: A guide to cutting your scientific calculations in smaller pieces.** TLC Publishing, 2015. Disponível em: <https://books.google.com.br/books?id=GeNkCQAAQBAJ>.

JESSHOPE, C.; EGAN, C. **Advances in Computer Systems Architecture: 11th Asia-Pacific Conference, ACSAC 2006, Shanghai, China, September 6-8, 2006, Proceedings**. Springer, 2006. (LNCS sublibrary: Theoretical computer science and general issues). ISBN 9783540400561. Disponível em: <https://books.google.com.br/books?id=0IY7LW5J4JgC>.

KARLIN, I.; KEASLER, J.; NEELY, R. **LULESH 2.0 Updates and Changes**. [S.l.], 2013. 1-9 p.

KOWARSCHIK, M.; WEISS, C. An overview of cache optimization techniques and cache-aware numerical algorithms. **Algorithms for Memory Hierarchies**, Springer, p. 213–232, 2003.

KRAUSE, A. M.; MORO, G. B.; SCHNORR, L. M. Análise de Desempenho da Multiplicação de Matrizes por Strassen contra o Método Tradicional. XIV Workshop de Processamento Paralelo e Distribuído, p. 14–17, 2016.

LAURENZANO, M. A. et al. Reducing energy usage with memory and computation-aware dynamic frequency scaling. In: SPRINGER. **European Conference on Parallel Processing**. [S.l.], 2011. p. 79–90.

LI, D. et al. Strategies for energy-efficient resource management of hybrid programming models. **IEEE Transactions on parallel and distributed Systems**, IEEE, v. 24, n. 1, p. 144–157, 2013.

LI, Y.; LAN, Z. A survey of load balancing in grid computing. **Computational and Information Science**, p. 280–285, 2005. ISSN 03029743. Disponível em: <http://www.springerlink.com/index/4m49u0mh1l8bntgk.pdf>.

LIM, M. Y.; FREEH, V. W.; LOWENTHAL, D. K. Adaptive, transparent cpu scaling algorithms leveraging inter-node mpi communication regions. **Parallel Computing**, Elsevier, v. 37, n. 10, p. 667–683, 2011.

LLNL. **Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory**. [S.l.], 2018. 1-17 p.

LORENZ, D. et al. A comparison between opari2 and the openmp tools interface in the context of score-p. In: SPRINGER. **International Workshop on OpenMP**. [S.l.], 2014. p. 161–172.

MILLANI, L. F.; SCHNORR, L. M. Computation-aware dynamic frequency scaling: Parsimonious evaluation of the time-energy trade-off using design of experiments. In: SPRINGER. **European Conference on Parallel Processing**. [S.l.], 2016. p. 583–595.

MOLKA, D. et al. Detecting memory-boundedness with hardware performance counters. In: ACM. **Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering**. [S.l.], 2017. p. 27–38.

MORO, G. B.; SCHNORR, L. M. Measuring hardware counters for hpc application phase detection. XIV Workshop de Processamento Paralelo e Distribuído, p. 37–39, 2016.

MORO, G. B.; SCHNORR, L. M. Detecting memory-bound parallel regions to improve the energy-efficiency of applications. XV Workshop de Processamento Paralelo e Distribuído, p. 10–13, 2017.

ORGERIE, A.-C.; ASSUNCAO, M. D. de; LEFEVRE, L. A Survey on Techniques for Improving the Energy Efficiency of Large-scale Distributed Systems. **ACM Comput. Surv.**, v. 46, n. 4, p. 47:1–47:31, 2014. ISSN 0360-0300.

PACHECO, P. **An Introduction to Parallel Programming**. Elsevier Science, 2011. (An Introduction to Parallel Programming). ISBN 9780080921440. Disponível em: <https://books.google.com.br/books?id=SEmfraJjvfwC>.

PADOIN, E. L. et al. Using power demand and residual load imbalance in the load balancing to save energy of parallel systems. **Procedia Computer Science**, Elsevier, v. 108, p. 695–704, 2017.

SHAFIK, R. A. et al. Adaptive energy minimization of openmp parallel applications on many-core systems. In: ACM. **Proceedings of the 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures**. [S.l.], 2015. p. 19–24.

SILVEIRA, D. S. et al. Energy Consumption Estimation in Parallel Applications: an Analysis in Real and Theoretical Models. WSCAD 2016 - XVII Simpósio em Sistemas Computacionais de Alto Desempenho, p. 134–145, 2016.

SUBRAMANIAN, L. Providing high and controllable performance in multicore systems through shared resource management. **arXiv preprint arXiv:1508.03087**, 2015.

SUEUR, E. L.; HEISER, G. Dynamic voltage and frequency scaling: The laws of diminishing returns. 2010.

SUEUR, E. L.; HEISER, G. Dynamic voltage and frequency scaling: the laws of diminishing returns. **Proceedings of the 2010 international conference on Power aware computing and systems**, p. 1–8, 2010.

WANG, W. et al. Using per-loop cpu clock modulation for energy efficiency in openmp applications. In: IEEE. **Parallel Processing (ICPP), 2015 44th International Conference on**. [S.l.], 2015. p. 629–638.

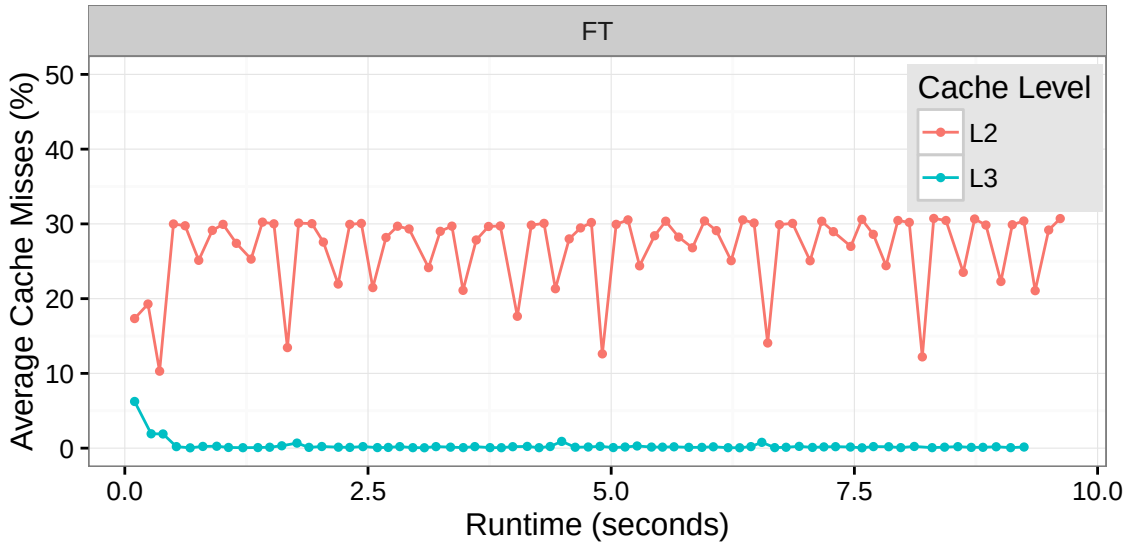
APÊNDICE A — EXPERIMENTOS AUXILIARES COM *WORKFLOW* PROPOSTO

Esse apêndice apresenta alguns experimentos realizados utilizando as ferramentas Likwid e ScoreP, na etapa de coleta de contadores de hardware da fase 1 do *Workflow* proposto.

A.1 Utilizando Likwid

O experimento tem por objetivo coletar alguns contadores de hardware para identificar os trechos de execução mais limitados pela memória. Dentre as aplicações testadas, selecionamos a aplicação 3D *Discrete Fast Fourier Transform* (FT) (classe B) do NAS *Parallel Benchmark*. A plataforma utilizada no experimento é uma máquina com 2 processadores Intel (R) Xeon (R) E5-2650 CPU 2.00 GHz, cada processador possui 8 núcleos físicos, com suporte a tecnologia *Hyper-Threading*. A Figura A.1 apresenta a taxa de *misses* para as *Caches* L2 e L3 da aplicação 3D *Fast Fourier Transform*, a partir desse resultado é possível correlacionar as duas taxas de *misses*.

Figura A.1: Taxa de *Misses* para as *Caches* L2 e L3 - (NPB-FT, classe B) - Medição a cada 100 milisegundos (MORO; SCHNORR, 2016).

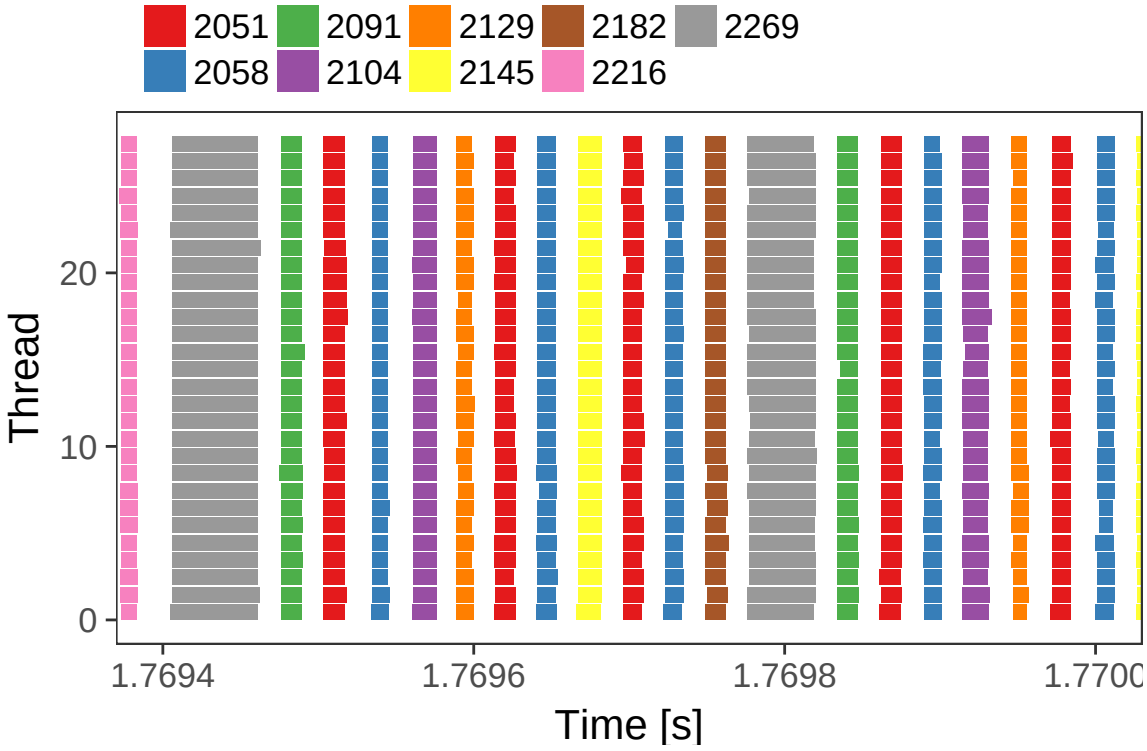


O maior resultado de *misses* obtido para o nível de *Cache* L3 é de 8%, os demais valores estão próximos a 0%. Já para o nível de *Cache* L2 é possível visualizar uma tendência, definindo fases em que a aplicação possui uma taxa de *misses* na faixa de 30%, intercaladas por fases de valores a baixo de 20% ou entre 15% a 20%. Com isso, é possível definir os trechos de execução mais limitados pela memória. Nessa abordagem mapeamos apenas marcas temporais, completamente relacionadas com o tamanho de entrada, plataforma alvo e condições que se replicadas, poderiam nos levar a resultados completamente distintos.

A.2 Utilizando ScoreP

O objetivo principal desse experimento é mapear as regiões paralelas mais limitadas pela memória, de forma que seja possível correlacionar as métricas obtidas com o código-fonte da aplicação alvo. A aplicação utilizada nesse experimento foi a Lulesh, utilizando como entrada os parâmetros "-s 15 -i 100". As execuções foram realizadas em uma plataforma de 2 processadores Intel(R) Xeon(R) CPU E5-2650 2.3GHz, com 20 núcleos físicos e suporte a *Hyper-Threading*. As Figuras A.2 e A.3 apresentam os principais resultados obtidos.

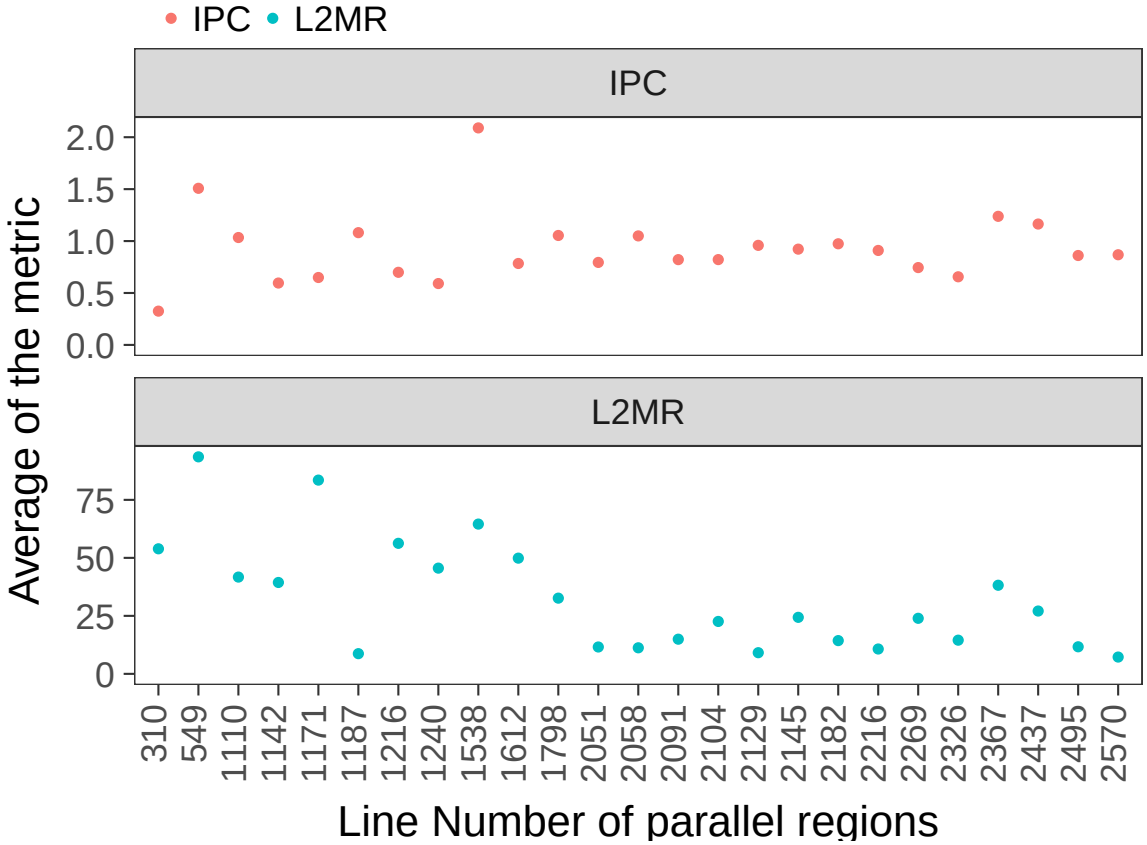
Figura A.2: Comportamento da Aplicação Lulesh por linha de código (MORO; SCHNORR, 2017).



A Figura A.2 apresenta um demonstrativo geral das regiões paralelas por *thread* de processamento, cada linha do gráfico representa uma *thread*, cada coluna representa uma região paralela identificada. O nome das regiões paralelas é definido a partir da linha inicial da respectiva região. A partir desse gráfico podemos mapear a ordem de execução das regiões, como também a duração de cada uma delas.

Na Figura A.3 é possível visualizar uma relação entre as medidas de IPC e taxa de *misses* da *Cache* L2 por região paralela. Cada ponto da figura apresenta uma coleta realizada na *thread* 0 da aplicação, mas de forma comparativa, as demais 27 *threads* utilizadas compartilham o mesmo comportamento. Dentre os resultados, é possível destacar que existem pontos aonde a taxa de *misses* na *Cache* L2 superou os 50% e outros pontos que a medida de IPC obteve 2 (maior valor). Alguns casos, como na região 2058, é possível destacar um comportamento mais *CPU-Bound*, pois existe uma baixa taxa de misses, cerca de 25% e um IPC a cima de 1. Já na região 310, por exemplo, ocorre um comportamento diferente, uma taxa de *misses* de 50% e um IPC a baixo de 0,5, definindo um comportamento mais limitado pela memória (*Memory-Bound*).

Figura A.3: IPC e Taxa de *Misses* na *Cache* L2 por linha de código - Lulesh (MORO; SCHNORR, 2017).



APÊNDICE B — UTILIZAÇÃO DA LIBNINA

Nesse apêndice são apresentados detalhes práticos sobre utilização da LibNina, utilizando como exemplo a aplicação Lulesh. A LibNina está disponível no repositório <https://github.com/tido4410/libnina>. Como a biblioteca utiliza contadores de hardware para investigar as regiões paralelas, é necessário a instalação da biblioteca PAPI (*Performance Application Programming Interface*). Já para a mudança de frequência, devemos instalar a biblioteca CPUFreq. Além disso, é necessário que o sistema suporte a política *Userspace*, pois a biblioteca altera a frequência das regiões paralelas, utilizando essa política. A LibNina utiliza o compilador (*source-to-source*) Opari2, essa ferramenta é utilizada para gerar a partir do código fonte da aplicação, um código que realizará chamadas à LibNina. A seguir é possível visualizar um exemplo de *script* que realiza de forma simplificada a instalação dos pré-requisitos.

Código B.1 – Exemplo de Instalação de Pré-Requisitos.

```
sudo apt-get install libpapi-dev -y
sudo apt-get install libconfig-dev -y
sudo apt-get install libcpufreq-dev -y
sudo apt-get install opari2
```

Após a configuração do ambiente, é necessário realizar a compilação da aplicação com a ferramenta Opari2. A seguir é apresentado um exemplo de *Makefile,* criado especialmente para a aplicação Lulesh.

Código B.2 – Makefile para Aplicação Lulesh.

```
SHELL = /bin/sh
.SUFFIXES: .cc .o
LULESH_EXEC = lulesh2.0
BINDIR = /opt/opari2/bin
NM = `$(BINDIR)/opari2-config -nm`
AWK_SCRIPT = `$(BINDIR)/opari2-config --region-initialization`
SERCXX = g++ -DUSE_MPI=0
CXX = $(SERCXX)
SOURCES2.0 = \
        lulesh.mod.cc \
        lulesh-comm.cc \
        lulesh-viz.cc \
        lulesh-util.cc \
        lulesh-init.cc
OBJECTS2.0 = $(SOURCES2.0:.cc=.o) pompregions.o
OPARI2FLAGS= $($NM) --cflags)
CXXFLAGS = -g -O3 -fopenmp $(OPARI2FLAGS) -I. -Wall
LDFLAGS = -g -O3 -L/home/gabrielbmoro/svn/libnina/src -lnina -
   fopenmp

.cc.o: lulesh.h
        @echo "Building␣$<"
        $(CXX) -c $(CXXFLAGS) -o $@ $<
.c.o:
        @echo "Building␣$<"
        $(CC) $(POMP_INC) -c $? -o $@
all: $(LULESH_EXEC)
```

```
pomp2regions.c: lulesh.mod.o
        $(NM) lulesh.mod.o | $(AWK_SCRIPT) > pompregions.c
lulesh2.0: $(OBJECTS2.0)
        @echo "Linking"
        $(CXX) $(OBJECTS2.0) $(LDFLAGS) $($(OPARI2FLAGS) --
           cflags) -lm -o $@

clean:
        /bin/rm -f *.o *~ $(OBJECTS) $(LULESH_EXEC)
```

B.1 Execução da LibNina em Modo PAPI

Para executar a LibNina coletando contadores de hardware, primeiramente é necessário definir quais contadores de hardware serão utilizados, modificando o arquivo modeling-libnina.cfg (localizado na pasta "src" da LibNina). O arquivo é apresentado a seguir:

Código B.3 – Arquivo de Definição dos Contadores de Hardware.

```
filename = "libnina"
papi: {
   counters = ( "PAPI_TOT_CYC", "PAPI_TOT_INS" );
}
```

Com os contadores especificados, é possível executar a LibNina em Modo PAPI, utilizando os seguintes comandos:

Código B.4 – Script para Executar a Aplicação Lulesh com LibNina em Modo PAPI.

```
export NINA_PAPI=true
export OMP_NUM_THREADS=20
sudo -E LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/gabrielbmoro/svn/
   libnina/src ./lulesh2.0
```

A execução da LibNina em Modo PAPI gera um arquivo CSV que define para cada região paralela: a linha de código, *timestamp* de início, *timestamp* de fim, duração, quantidade de ciclos (primeiro contador), quantidade de instruções (segundo contador), e assim por diante.

B.2 Execução da LibNina com Modo de Seleção de Frequência

A LibNina pode ser executada em modo de seleção de frequência, para isso precisamos definir um arquivo de configuração, atribuindo uma frequência para cada região paralela. Cada linha desse arquivo deverá definir a linha de código, a frequência em MHz e o nome do arquivo de código da respectiva região. A seguir é possível visualizar um exemplo de *script* que executa a aplicação Lulesh utilizando o modo de seleção de frequência.

Código B.5 – Execução da Aplicação Lulesh com a LibNina com Seleção de Frequência.

```
ninafile=$expdatapath/nina_file.conf
export NINA_AMOUNT_OF_CPUS=20
export NINA_TARGET_CPUS
   =0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19
```

```
export NINA_CONFIG="$ninafile"
unset NINA_PAPI
export OMP_NUM_THREADS=20
export OMP_PROC_BIND=true

sudo -E LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/svn/libnina/src
   ./lulesh2.0
```

Mais detalhes podem ser visualizados no repositório da dissertação: <https://bitbucket.org/gbmoro/dissertacao_gbmoro>.